Une semantique formelle du P-Code basee sur les types abstraits algebriques

Joëlle Despeyroux-Savonitto

IRIA
CENTRE DE ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
BP 105
78153 Le Chesnay Cedex
France
Tél. 954 90 20

Rapports de Recherche

N° 158

UNE SÉMANTIQUE FORMELLE DU P-CODE BASÉE SUR LES TYPES ABSTRAITS ALGÉBRIQUES

Joelle DESPEYROUX-SAVONITTO

Septembre 1982

UNE SEMANTIQUE FORMELLE DU P-CODE

BASEE SUR LES TYPES ABSTRAITS ALGEBRIQUES

Joelle DESPEYROUX-SAVONITTO

Résumé

Cet article présente une sémantique formelle du P-Code, langage d'assemblage relativement puissant.

Le P-Code choisi est un P-Code très proche de celui du système UCSD. C'est la dernière version du P-Code UCSD, modifiée non fondamentalement, pour des raisons de clarté.

La sémantique est donnée à l'aide des types abstraits algébriques, selon la méthode décrite dans la thèse de M.C. Gaudel, rappelée ici de manière très succinte.

L'originalité de ce travail consiste en la donnée d'une sémantique formelle d'un langage de bas niveau.

Mots clés : Types abstraits - P-Code -

Abstract

This report present a semantics of P-Code, a low-level language.

The P-Code defined here is very closed to the last version (IV.0) of P-Code in system UCSD. The semantics used is based on algebraic abstract data types.

Key Words : Algebraic Abstract data types - P-Code -

P L A N

I. - INTRODUCTION

Le langage Pascal étant créé, il se posait le problème d'en écrire un compilateur. Pour en simplifier l'écriture, l'idée est venue de considérer une machine abstraite, la P-machine à pile, avec son code, le P-Code.

Le P-Code devait satisfaire à deux exigences : d'une part être facilement microprogrammable sur un émulateur, d'autre part être un langage intermédiaire correct, la génération de code pour une machine quelconque devant être aisée.

Le P-Code ainsi construit [JKM 74] a changé de l'implémentation effective sur une petite machine pour devenir le P-Code 4 [AJN 75].

Depuis, le Pascal a évolué et a été implémenté sur des machines totalement différentes. Le P-Code a alors donné lieu à deux familles de P-Code [Nel 79]. L'une représentée par le P-Code LASL [Mor 76], l'autre par le P-Code UCSD [USE 81].

Le P-Code LASL devait être un langage intermédiaire pour traduire du Model, nette extension du Pascal, en code machine pour le CRAY-1. Le critère essentiel à satisfaire étant la rapidité, le P-Code dû être notablement étendu, utilisant au mieux les registres du CRAY, permettant le retour de plusieurs valeurs ...

Le P-Code UCSD devait être facilement émulable sur une petite machine et traduire du Pascal UCSD, extension du Pascal "standard" [JW 78].

Le critère était surtout le gain de place. La définition de la P-machine fut rigidifiée, et le P-Code devint un peu plus puissant et compact, par ajout d'instructions.

Devant cette multitude de P-Code, que choisir ?

Notre propos ici est de donner un type abstrait décrivant un P-Code afin de générer automatiquement, grâce au système PERLUETTE [Des 80] (basé sur une méthode utilisant les types abstraits algébriques décrite dans [Gau 80]), un compilateur Pascal (Pascal ISO [Iso 81])→ PCode.

Le P-Code choisi est le P-Code UCSD, à peine modifié pour satisfaire à la fois au critère de réalisme (le traducteur produit correspondant à une traduction type rencontrée dans le monde réel) et de clarté (le P-Code est "épuré" des instructions qui n'ajoutent rien à la complexité du langage. Le langage est utilisé sous la forme assembleur et non langage machine (le code étant intouchable par les instructions qui sont considérées "à part, dans la mémoire", les sauts se faisant à des étiquettes et non à des adresses fixes de la mémoire)).

Le Pascal UCSD diffère du Pascal ISO (le Pascal "normalisé") notablement par le traitement des chaînes de caractères, les entrées-sorties et la possibilité de compilation séparée et de "co-routines". La P-machine choisie et son code différent en conséquence de ceux du système UCSD. La liste complète des différences est donnée en annexe (A.1).

Remarque importante

Il faut bien noter que le P-Code n'est pas à proprement parler un langage étant donné que les instructions ont un "sens" seulement par rapport à une P-machine donnée (une certaine organisation des données en mémoire, un système, et le P-Code).

Il faut ainsi renoncer à l'idée d'un "langage" manipulant une pile qui contiendrait des "enregistrements" pour chaque "procédure", des "entiers", des "ensembles", et ferait des appels et retour de "procédure".

Le P-Code manipule des mots, ou ensembles de mots, qu'il trouve à des adresses dans une mémoire. Les instructions sont bien faites pour traduire des traitements d'entier, d'ensemble ou des appels et retour de procédure, mais rien n'empêche d'additionner, comme "entiers", deux mots mis dans la mémoire en tant que partie d'un ensemble, ou d'écraser une étiquette retour de procédure ...

II. - DESCRIPTION DU P-CODE CHOISI

Nous donnons tout d'abord "l'organisation mémoire" de la P-machine, car sa connaissance est nécessaire à la compréhension des instructions. Par "organisation mémoire" nous entendons l'implantation des données en mémoire lors de l'exécution d'un programme P-Code correct.

Afin de faciliter la lecture des instructions, nous conseillons au lecteur de garder l'"image mémoire" en tête, sinon à portée de main.

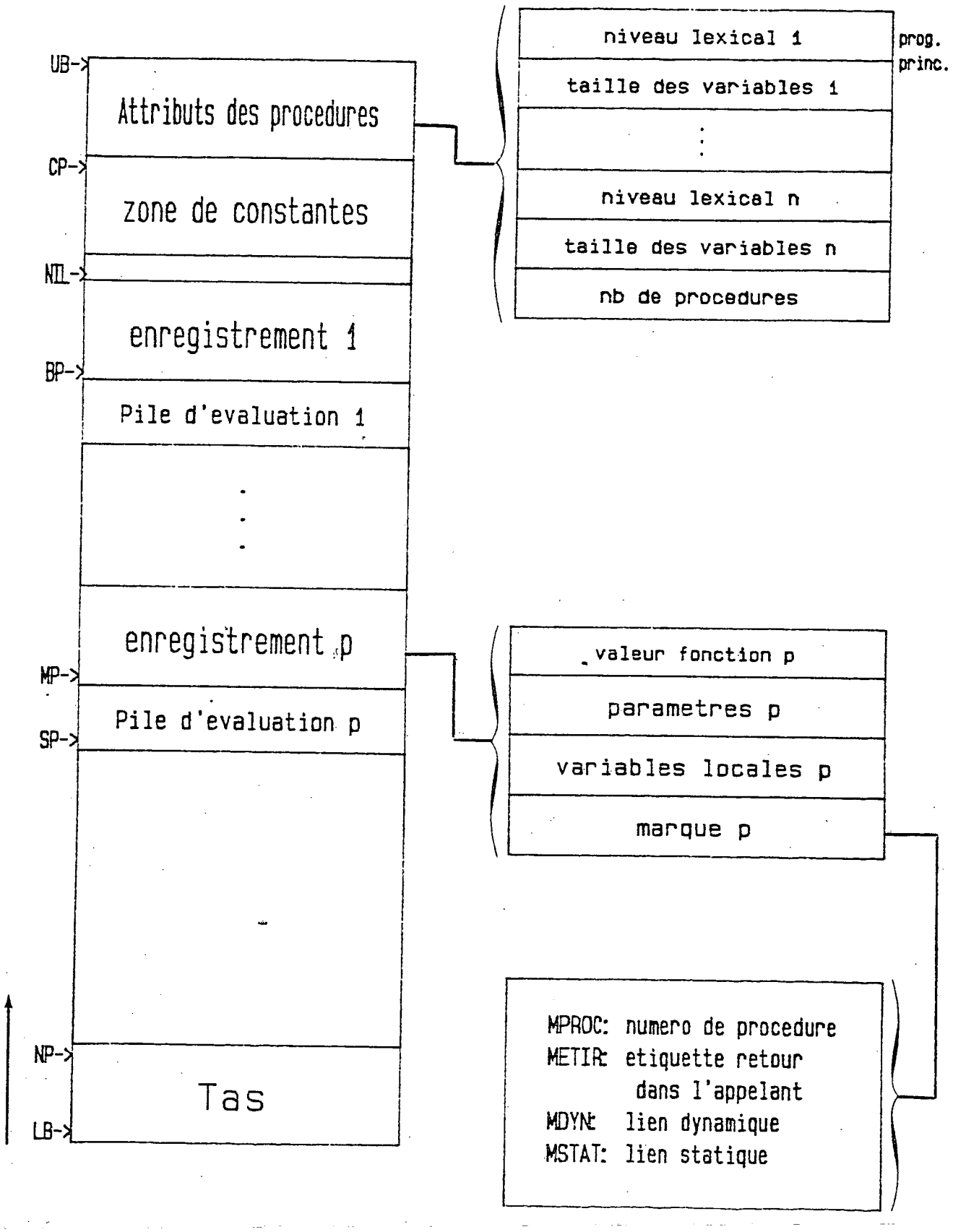
. La zone mémoire est vue sous la forme de deux piles s'étendant dans le même espace libre (l'une croissant vers le haut, l'autre vers le bas). La première contient une zone pour les attributs des procédures du programme, une zone de constantes, puis une pile d'enregistrements et de piles d'évaluation par procédure en cours d'exécution. Le rôle de ces différentes zones est détaillé ci-dessous.

- Les attributs de procédure sont, pour chaque procédure et pour le programme principal, son niveau lexical et le nombre de mots occupés par ses variables. Cette zone se termine par le nombre de procédures du programme (lui-même compris).

- La zone de constantes contient les constantes de plus d'un mot (ex. : les réels).

- L'enregistrement d'une procédure contient un ou deux mots pour le retour de la fonction si c'en est une, une zone qui contient les paramètes, une autre les variables locales, et une troisième la marque. La marque permet de dépiler l'enregistrement d'une procédure lors de son retour (laissant éventuellement la valeur de la fonction "dans" la pile d'évaluation de la procédure appelante) et de faire appel aux variables globales. La marque contient MPROC, le numéro de la procédure, METIR, l'étiquette retour dans la procédure appelante, MDYN, pointeur vers l'enregistrement de l'appelant, MSTAT, pointeur vers le père lexical de la procédure.

IMAGE MEMOIRE



- enfin la pile d'évaluation contient les calculs intermédiaires.

Des pointeurs (adresses) repèrent ses différentes zones. La mémoire est contenue entre LB (Lower-Bound) et UB (Upper-Bound). NP (New-Pointer) pointe vers le dernier mot du tas. NIL désigne une adresse licite dont le contenu ne l'est pas. CP (Constant-Pointer) désigne le premier mot (dans le sens croissant de la pile) de la zone de constante, BP (Base-Pointer) désigne le haut de l'enregistrement du programme, MP (Mark-Pointer) désigne le haut de l'enregistrement courant (MSTAT), et SP désigne le haut de la pile d'évaluation courante, soit le haut de la seconde pile (son adresse la plus basse dans la mémoire).

Les instructions du P-Code peuvent être groupées selon leur utilité :

- rangement de constante,
- accès local, global ou intermédiaire selon l'enregistrement (la procédure accédée),
- accès indirect, accès multiple (concernant un groupe de mots),
- calcul d'adresse (utilisé pour l'indexation dans un tableau, par exemple),
- opération sur booléens, entiers, réels ou ensembles,
- branchement,
- appel et retour de procédure.

Chaque instruction est donnée par son mnémonique, ses paramètres (W pour mot, B pour octet), ce qu'elle prend et rend sur la pile d'évaluation courante et sa signification.

Selon le type de l'instruction, on considère que l'on a sur la pile d'évaluation (cf remarque p. I3) :

- une étiquette (eti)
- un mot, une adresse (notée adr), l'adresse nil, un bloc de mots (bloc),
- un ensemble (ens), c'est-à-dire un bloc de mots représentant cet ensemble suivi d'une valeur qui est sa taille (nombre de mots),
- un booléen (bool ; mot de la forme 0...0 suivi de 0 ou 1), un entier (ent ; mot représentant un entier par complémentation à 2), un réel (2 mots représentants un flottant), un enregistrement (enreg), des paramètres (param , un bloc de mots), 0, 1 ou 2 mots contenant la valeur retour d'une fonction (fnc).

La signification d'une instruction est donnée en français, (par un terme du type abstrait), par dessin, ou relativement à une autre instruction.

Dans cette partie le type abstrait n'est utilisé que comme aide à la compréhension d'une instruction.

. Notations utilisées

Si a désigne une adresse, [a] désigne le contenu-mémoire trouvé à cette adresse. Si a est une adresse et m une valeur a + m désigne l'indexation de a par m. [a] ← b signifie qu'à l'adresse a on trouve désormais le mot b. (le lecteur a sans doute compris qu'un mot était soit une adresse, soit une valeur (entier, booléen...)).

empiler (a) signifie : SP ← SP $\ominus$ 1[(1)], [SP] ← a ; (ceci pour chaque mot de l'ensemble si a est un ensemble)

dépiler signifie : SP ← SP $\oplus$ 1 ;

dépiler-2 signifie dépiler 2 fois

load 7 désigne l'instruction de mnémonique load et de paramètre le mot représentant l'entier 7.

cont-ens (a,b) désigne le bloc de mots contenu entre les adresses a et b, "vu comme" un ensemble.

{x..y} est l'ensemble qui contient les valeurs comprises entre x et y (compris).

---

(1) La pile d'évaluation croît vers les adresses décroissantes.

Ainsi LDCI est décrite par :

LDCI W ( ) → mot empiler(W)

Car LDCI prend un paramètre (mot) W, ne prend rien sur la pile et y met un mot, et sa signification est empiler(W).

SLDC7 est décrite par :

SLDC7 ( ) → mot LDCI 7

Car SLDC7 à la même signification que LDCI 7, mais n'a pas de paramètre.

Description des instructions groupée selon leur "signification" :

• Chargement de constante :

Instruction	Paramètres	Prend et rend sur la pile	Signification
LDCI	W	( ) → mot	empiler(W) "chargement immédiat"
SLDCØ ⋮ SLDC31		( ) → mot ( ) → mot	LDCI Ø (instruction compacte) LDCI 31
LDCN		( ) → nil	empiler(nil)
LDCB	B	( ) → mot	LDCI B (où B est considéré comme une Valeur qui doit être comprise entre 'valeur-octet-maximale' et 'valeur-octet-minimale'
LAC	W	( ) → adr	empiler(CP ⊖ W)

Accès local :

Instruction	paramètre	prend et rend sur la pile	Signification
LDL	W	( ) → mot	empiler ([MP ⊕ W])
SLDL1		( ) → mot	LDL1
⋮			
SLDL16		( ) → mot	LDL16
LLA	W	( ) → mot	empiler (MP + W)
STL	W	(mot) →	[MP ⊕ W] → [SP] ; dépiler.

Accès global :

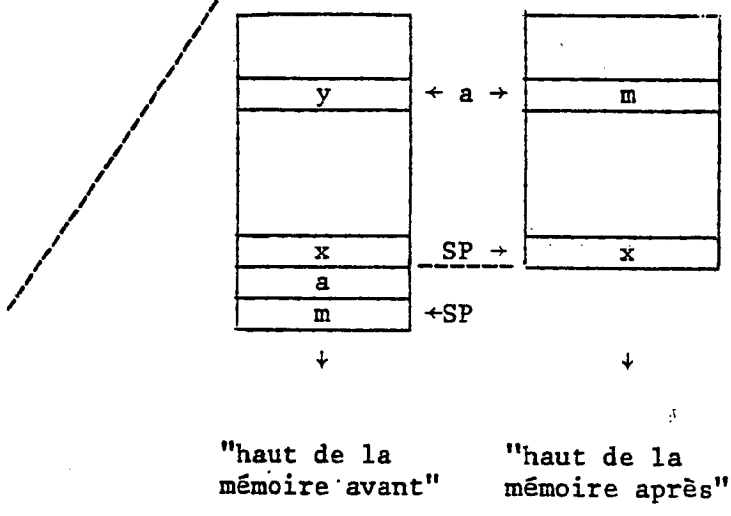
LDO	W	( ) → mot	empiler ([BP ⊕ W])
SLDO1		( ) → mot	LDO 1
⋮			
SLDO16		( ) → mot	LDO 16
LAO	W	( ) → adr	empiler (BP ⊕ W)
SRO	W	(mot) →	[BP ⊕ W] ← [SP] ; dépiler

Accès intermédiaire :

Instruction	Paramètre	prend et rend sur la pile	Signification
LOD	B W	( ) → mot	empiler ([enreg-ancêtre(B) ⊕ W]) enreg-ancêtre(B) est l'adresse de l'enregistrement situé à B liens statique de l'enregistrement courant. "accès à une variable globale"
LDA	B W	( ) → adr	empiler(enreg-ancêtre(B) ⊕ W)
STR	B W	(mot) →	[enreg-ancêtre(B) ⊕ W] ← [SP] ; dépiler

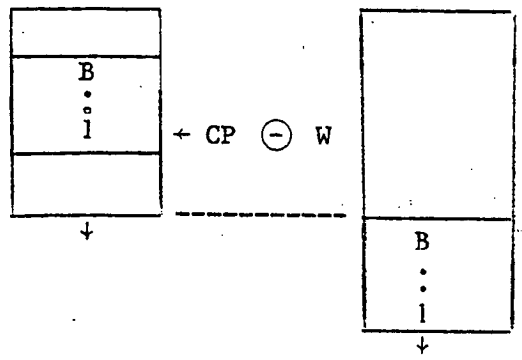
Accès indirect :

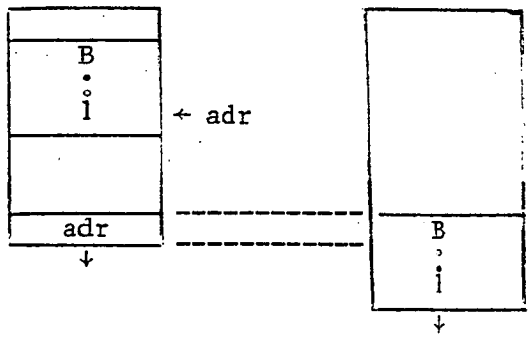
IND	W	(adr) → mot	[SP] ←[[SP] ⊕ W]
SINDO • • • SIND7		(adr) → mot (adr) → mot	IND 0 IND 7
STO		(adr,mot) →	[[SP ⊕ 1]] ← [SP] ; dépiler-2

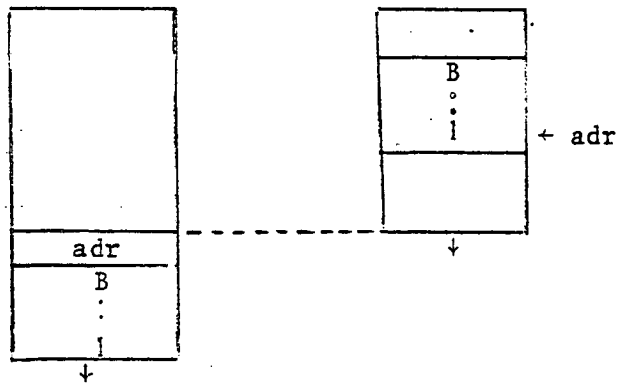


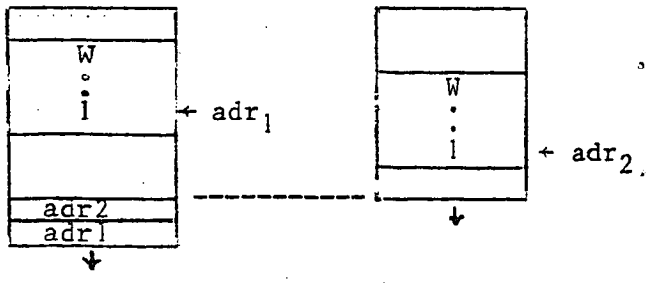
accès multiple

LDC	W B	( ) → bloc de mots
LDM	B	(adr) → bloc
STM	B	(adr,bloc)→
MOV	W	(adr_2,adr_1)









calcul d'adresse

INC	M	(adr) → adr	[SP] ← [SP] ⊕ M
IXA	M	(adr,mot) → adr	[SP ⊕ 1] ← [SP ⊕ 1] ⊕ ([SP] * M) ; dépiler ;

opérations sur les booléens

LAND		(mot,mot) → mot	[SP ⊕ 1] ← et ([SP],[SP ⊕ 1]) ; dépiler ;
LOR		(mot,mot) → mot	[SP ⊕ A] ← ou ([SP],[SP ⊕ 1]) ; dépiler ;
LNOT		(mot) → mot	[SP] ← non ([SP]) "complement à 1"
BNOT		(bool) → bool	[SP] ← non-bool ([SP]) "un booléen est 0--0 i ou i est 0 ou 1"

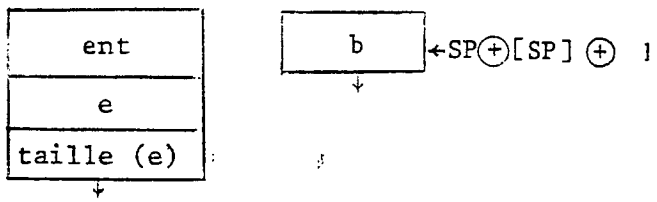
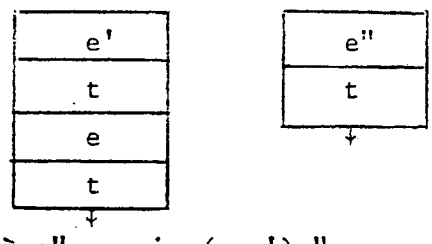
Opération sur les entiers

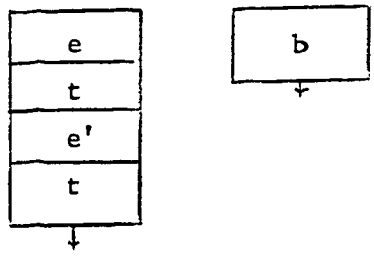
ABI	(ent) → ent	si [SP] ≠ ent '-32768' alors [SP] ← abs ([SP]) sinon erreur
NGI	(ent) → ent	si [SP] ≠ ent '-32768' "complément à 2" alors [SP] ← opp ([SP]) sinon erreur
INCI	(ent) → ent	[SP] ← incr ([SP])
DECI	(ent) → ent	[SP] ← decr ([SP])
ADI	(ent,ent) → ent	[SP ⊕ 1] ← add([SP ⊕ 1],[SP]) ; dépiler ;
SBI	(ent,ent) → ent	[SP ⊕ 1] ← sous([SP ⊕ 1],[SP]) ; dépiler ;
MPI	(ent,ent) → ent	[SP ⊕ 1] ← mult([SP ⊕ 1],[SP]) ; dépiler ;
DVI	(ent,ent) → ent	si [SP] ≠ ent 'o' alors [SP ⊕ 1] ← div ([SP ⊕ 1], [SP]) ; dépiler ; sinon erreur
MODI	(ent,ent) → ent	si [SP] > ent 'o' alors [SP ⊕ 1] ← mod([SP ⊕ 13, [SP]) dépiler sinon erreur
CHK	(ent,ent,ent) → ent	si [SP ⊕ 2] ∈ {[SP ⊕ 1]..[SP]} : alors dépiler-2 fois sinon erreur
EQUI	(ent,ent) → bool	[SP ⊕ 1] ← eg ([SP ⊕ 1], [SP]) ; dépiler ;
NEQI	(ent,ent) → bool	[SP ⊕ 1] ← non-eg ([SP ⊕ 1], [SP]) dépiler ;
LEQI	(ent,ent) → bool	[SP ⊕ 1] ← inf-eg ([SP ⊕ 1],[SP]) dépiler ;
LESI	(ent,ent) → bool	[SP ⊕ 1] ← inf ([SP ⊕ 1], [SP]) ; dépiler ;
GEQI	(ent,ent) → bool	[SP ⊕ 1] ← sup-eg ([SP ⊕ 1],[SP]) dépiler
GTRI	(ent,ent) → bool	[SP ⊕ 1] ← sup ([SP ⊕ 1], [SP]) ; dépiler ;

Opérations sur les réels

FLT	(ent) →réel	SP ← SP ⊖ 1 <SP> ← float([SP]) où < x > = cont-flottant(x)
TNC	(réel) → ent	[SP ⊕ 1] ← trunc(<SP>) ; dépiler
RND	(réel) → ent	[SP ⊕ 1] ← round(<SP>) ; dépiler
ABR	(réel) → réel	<SP> ← abs-flottant(<SP>)
NGR	(réel) → réel	<SP> ← opp-flottant(<SP>)
ADR	(réel,réel) → réel	<SP ⊕ 2> ← add-flottant(<SP ⊕ 2>, <SP>) ; dépiler-2 ;
SBR	(réel,réel) → réel	<SP ⊕ 2> ← sous-flottant(<SP ⊕ 2>,<SP>) ; dépiler-2 ;
MPR	(réel,réel) → réel	<SP ⊕ 2> ← mult-flottant(<SP ⊛ 2>,<SP>) ; dépiler-2 ;
DVR	(réel,réel) → réel	si <SP> ≠ réel 'o' alors <SP ⊕ 2> ← div-flottant(<SP ⊕ 2>,<SP>) ; dépiler-2 ; sinon erreur
EQUR	(réel,réel) → réel	<SP ⊕ 4> ← eg-flottant(<SP ⊕ 2>,< SP>) ; dépiler-2 ; dépiler
NEQR	(réel,réel) → réel	<SP ⊕ 4> ← non-eg-flottant(<SP ⊕ 2>,<SP>) ; dépiler-2 ; dépiler
LEQR	(réel,réel) → réel	<SP ⊕ 4> ← inf-eg-flottant(<SP ⊕ 2>,<SP>) ; dépiler-2 ; dépiler
LESR	(réel,réel) → réel	<SP ⊕ 4> ← inf-flottant(<SP ⊛ 2>,<SP>) ; dépiler-2 ; dépiler
GEQR	(réel,réel) → réel	<SP ⊕ 4> ← sup-eg-flottant(<SP ⊕ 2>,<SP>) ; dépiler-2 ; dépiler
GTRR	(réel,réel) → réel	<SP ⊛ 4> ← sup-flottant(<SP ⊕ 2>,<SP>) ; dépiler-2 ; dépiler

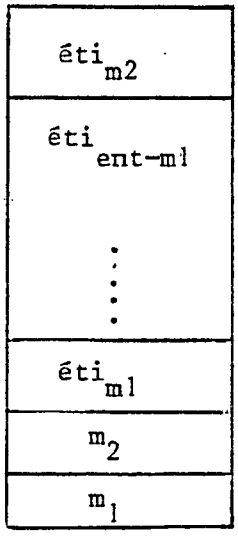
Opérations sur les ensembles :

ADJ	B	(ens) → bloc	si B = [SP] alors dépiler "(la valeur indiquant la taille de l'ensemble)" sinon dépiler ; empiler (B- SP) mots de valeur 'o'.
SRS		(ent,ent) → ens	si [SP ⊕ 1] > [SP] alors [SP ⊕ 1] ← ∅ ; dépiler ; "ensemble vide" sinon e ← {[SP ⊕ 1] .. [SP]} ; dépiler-2 ; empiler(e) ; empiler(taille(e)) erreur si ent ∈ [0 .. 4079] "e est une partie d'un ensemble qui sera complété par ADJ"
INN		(ent,ens) → bool	inn :  où b = in(ent,e) = in([SP ⊕ [SP] ⊕ 1], {SP ⊕ 1, SP ⊕ [SP]}) = si ent ∈ e alors vrai sinon faux.
UNI		(ens,ens) → ens	 où e" = union(e,e') "ou sur chaque mot"
INT		(ens,ens) → ens	défini de même que UNI en prenant l'intersection de e avec e', par "INT sur chaque mot".
DIF		(ens,ens) → ens	de même que UNI avec e" = diff(e',e) "e'_i et non e_i sur chaque mot e'_i et e_i"

EQUS	(ens,ens) → bool	 où b = eg-ens(e,e') "eg sur chaque mot"
NEQS	(ens,ens) → bool	de même que EQUS avec b = non-eg-ens (e,e')
LEQS	(ens,ens) → bool	de même que EQUS avec b = inf-eg-ens (e,e') "e inclus-dans e'"
GEQS	(ens,ens) → bool	"le contraire de LEQS "c'est-à-dire b = inf-eg-ens(e',e) = sup-eg-ens(e,e')

Branchements

UJP	"éti"	( ) →	saut inconditionnel à l'étiquette "éti". saut(eti)
FJP	"éti"	(bool) →	saut si faux à l'étiquette "éti". saut(eti)
TJP	"éti"	(bool) →	saut si vrai à l'étiquette "éti" .saut(eti)
NEQJ	"éti"	(ent,ent) →	saut si non-eg à l'étiquette "eti".saut(eti)
EQJ	"éti"	(ent,ent) →	saut si eg à l'étiquette "éti".saut(eti)
CJP	M	(ent) →	



Case(M)
= UJP éti ent-m1

erreur si ent ∉ [m1, m2]

"instruction case"

Case M = case-1 ([(CP ⊖ M) ⊕ ([SP] - [CP ⊖ M] + 1)])

Case-1(e) = dépiler ; saut(e)

avec :

[SP] ∉ {[CP ⊖ v]..[CP ⊖ v ⊕ 1]}

⟹ échec(case(v))

Procédures

N'oublions pas qu'il n'y a pas de procédure en P-Code mais des instructions bien faites pour traduire les appels et retours de procédures Pascal. Ceci dit, avant chaque appel de procédure :

- la pile contient éventuellement des calculs en cours, 0, 1 ou 2 mots pour réserver la place du retour de fonction et les paramètres.

A chaque appel de procédure, il faut :

- réserver la place nécessaire aux variables de la procédure appelée (celle-ci est donnée dans les attributs de la procédure).
- empiler la marque, c'est-à-dire le numéro de la procédure, l'étiquette retour, le lien dynamique et le lien statique).
- mettre à jour les pointeurs MP et SP.
- enfin, aller-à-la-procédure.

A chaque retour de procédure (la pile d'évaluation est vide et l'instruction de retour a comme paramètre n, le nombre de mots occupés par les variables locales et les paramètres) il faut :

- dépiler le nombre de mots nécessaires (n + 4, 4 étant la taille de la marque) pour ne laisser sur la pile que le résultat, s'il y en a.
- mettre à jour MP et SP.
- aller-à-l'étiquette-de-retour (qui est trouvée à MP $\ominus$ 1 (avant la mise à jour de MP)).

Procédures

CPL	B	(param) → enreg.	appel de la procédure local B : le lien statique ← MP
CPG	B	(param) → enreg.	appel de la procédure globale B : le lien statique ← BP
CPI	B_1 B_2	(param) → enreg.	appel de la procédure intermédiaire B_2, qui est à un niveau lexical inférieur de B_1 à celui de l'appelant : trouver l'enregistrement à B_1 liens statiques de l'enregistrement courant. Le lien statique ← le lien statique de cet enregistrement.
CPP	B	(param) → func	appel de la proc. prédéfinie B : le résultat éventuel est mis sur la pile (ex : new, write)
RPU	B	(enreg) → func	retour d'une proc : dépiler (B + 4) mots (4 est la taille de la marque)
autre :			
NOP			rien
DUPI		(mot) → mot mot	empiler ([SP]) "duplication du sommet de pile"
DUPR		(réel) → réel réel	empiler (<SP>)
SWAP		(mot,mot) → mot mot	"échange des contenus de SP et de SP ⊕ 1"

III. - LA METHODE UTILISEE

La méthode utilisée est décrite dans la thèse de M.C. Gaudel [Gau 80]. Notre propos est d'en extraire "un sous-ensemble minimal" permettant au "non-initié" de lire la suite avec profit.

Cette méthode, qui permet de donner une sémantique des langages de programmation (ainsi qu'une spécification d'un traducteur), utilise les types abstraits algébriques.

III.1. - Présentation d'un type abstrait

Nous serons très intuitifs dans cette partie.

Soulignons qu'il s'agit là d'expliciter des notations, sans entrer dans le monde des types abstraits.

La formalisation des concepts présentés pouvant être trouvée dans [Gau 80] mais aussi dans une littérature abondante, par exemple dans [GTW 78].

Un type abstrait est la donnée d'un ensemble de noms de types et d'opérations sur ces types. Les opérations sont données par leur profil et des axiomes qui spécifient leur propriétés.

Exemple :

```
Type logique
   Opérations :
            (logique) → logique : non
            (logique,logique) → logique : et, ou, implique
            ...
   Axiomes
     Vars :
            l1, l2 : logique
            implique (l1,l2) = non(l1) ou l2
            ...

Type entier
   Opérations :
            (entier,entier) → entier : add
            ...
   Axiomes :
     Vars :
        e : entier
            add(e,entier '0') = e
            ...
...
```

Notons que nous avons parfois besoin de constante dans un type (entier '0'). La notation utilisée suppose une opération prédéfinie qui à partir d'un texte (0) donne un terme du type (entier '0').

Le type abstrait, associé à un langage réel, contiendra toujours le type logique avec les constantes 'vrai' et 'faux'. En effet, pour définir l'égalité sur un type, il est nécessaire de disposer d'un type comportant deux objets différents : on pourra ainsi écrire

eg(entier '0', succ(entier '0')) = logique 'faux'

Le type logique est aussi nécessaire à la spécification des erreurs (précondition) comme nous allons le voir immédiatement.

III.2. - Les erreurs

Pour le cas des erreurs (division entière par 0 par exemple), plusieurs formalismes ont été développés jusqu'ici.

Nous avons choisi ici celui de Guttag, qui semble à l'heure actuelle le plus raisonnable.

Dans ce formalisme deux cas d'erreurs sont distingués ; les préconditions et les cas d'échec. On écrit

pre(div(x,y)) = non(eg(y,entier '0'))

ou bien

y = 0 ==> échec(div(x,y)).

La différence fondamentale entre ces deux types d'erreurs est que l'utilisateur du type abstrait doit vérifier les préconditions alors que c'est l'implémenteur du type abstrait qui devra vérifier les cas d'échec.

Ainsi toute précondition devra être vérifiée avant l'écriture du terme sur lequel elle porte, tandis que les cas d'échec seront traités par le système de la machine support.

Il existe deux sortes d'échec. L'un est l'échec "normal" : formule ==> échec ; si la formule est vraie, l'échec a lieu. L'autre est un échec optionnel : si la formule a lieu, l'échec peut arriver mais ce n'est pas obligatoire. Les deux cas sont illustrés dans le chapitre IV.

III.3. - Sémantique des langages

Considérons un langage évolué :

Un type abstrait le décrivant contiendra les types :

Types : logique, entier, var, tableau, ...

Le type var (var entière) contiendra l'opération :

op : (var) → entier : valeur.

A côté de ce type se trouve le type modif, dont les opérations correspondent aux instructions du langage.

op : (var, entier) → modif : affecter.

Pour définir la sémantique des instructions, nous avons besoin de la notion d'état.

De manière intuitive, un état est l'ensemble de toutes les formules vraies à un instant donné de l'exécution du programme.

Il contient par exemple :

type(var 'i') = entier
valeur(var 'i') = entier '0'

La signification d'un terme de type modif sera la modification d'un état. Par exemple si la formule

valeur(var 'x') = entier '0'

est valide dans l'état S_0

La signification du terme

affecter (var 'x', Succ(entier '0'))

sera le changement de l'état S_0 en l'état S_1 contenant maintenant la formule

valeur(var 'x') = succ(entier '0')

Les propriétés de l'opération valeur sont modifiées : la modification de l'état correspond à la transformation du type abstrait.

Pour décrire ce changement d'état, on utilise la substitution subst et une opération appl :

(modif, état) → état : appl

Les opérations du type modif sont définies en terme de ces primitives.

exemple :

appl(affecter(v,i),S)
= appl(subst(valeur(v),i),S)

où v est une variable, i un entier, et S un état.

Nous avons maintenant tous les éléments pour donner la sémantique d'un programme. Prenons un programme P. Sa valeur sémantique est ici un terme de type modif, m, séquence de déclaration, d'affectation, etc ...
L'état initial, SI, contient toutes les propriétés vraies avant l'exécution du programme.

Considérons maintenant l'application de cette modification m à l'état initial.

Appl(m,SI) se réécrit, par les définitions des modifs données dans le type abstrait, sous une forme appl(mp,SP), qui ne contient que les opérations appl, subst, et les opérations du type abstrait primitif (sans le type modif).

L'interprétation de appl(mp,SP), modifiant de proche en proche l'état initial, correspond à l'exécution du programme.

L'état résultat obtenu est la sémantique du programme.

Un exemple :

Le type abstrait :

Types : Entier, Booléen , Var, Scalaire ..., Modif
Opérations : sur les Entiers : Succ, Add, ...
sur les Var : Valeur : (Var) → Entier, Est-déclaré, Type, ...
sur les Modif : Decl-var, Affecter, ...
Axiomes : Add(x,Entier '0') = x ; ...
Définition des modifications : Appl(Decl-var(v,s),S)
= Appl(Subst(<type(v),Est-déclaré(v)>,
<Scalaire 'integer',
Logique 'vrai'>),
S)
Appl(Affecter(v,i),S)
= Appl(Subst(Valeur(v),i),S)

Soit le programme :

```
Program P ;
    var I : integer ;
    begin I : = 1 ;
    end.
```

L'état initial associé au type abstrait contient :

```
Add(Entier '1', Entier '0') = Entier '1' ...
Est-déclaré(Var 'a') = Logique 'faux' ...
```

(ceci pour toute constante (entière ou variable) possible. une opération : Text → type est sous-entendue pour tout type comportant des constantes)

...

Le terme associé à P est

```
S(P) = seq(Decl-var(Var 'I', Scalaire 'integer'),
           Affecter(Var 'I', Succ(Entier '0')))
```

La sémantique de P est l'état final obtenu en appliquant S(P) à l'état initial : Appl (S(P),I).

Cette fonction se réécrit, grâce aux définitions des modifications, pour n'obtenir que des termes primitifs, Subst et Appl :

```
  Appl(S(P),I)
= Appl(Affecter(Var 'I',Succ(Entier '0')),
       Appl(Decl-var(Var 'I', Scalaire 'integer')),
            I).

= Appl(Subst(Valeur(Var 'I'),Succ(Entier '0')),
       Appl(Subst(<Type(Var 'I'),Est-déclaré(Var 'I')>,
                  <Scalaire 'intéger', Logique 'vrai'>),
            I))
```

L'état résultat contient

- les même formules "générales" que l'état initial :

  Add(Entier '1', Entier '0') = Entier '1', ...

  ...

- des formules en plus :

  Valeur(Var 'I') = Succ(Entier '0')

  Est-déclaré(Var 'I') = Logique 'vrai'

  ...

C'est cette méthode que nous allons utiliser pour donner une sémantique formelle du P-Code.

IV. - PRESENTATION DU TYPE ABSTRAIT P-CODE

Le type abstrait complet se trouve en annexe (A.2) car il est relativement long. Le but de ce chapitre est d'en faire ressortir les points importants. Après avoir donné (IV.1. -) les principaux types et opérations, avec quelques axiomes munis des préconditions et cas d'échec qui leur sont liés, nous exposerons (IV.2. -) les principales options et problèmes apparus (une ou deux idées introduites de façon intuitive en IV.1. - seront explicitées en IV.2. -).

IV.1. - Les principaux éléments du type abstrait

Les types (ou sortes) de ce type abstrait sont

a) les types primitifs

- le type logique, nécessaire dans le type abstrait de tout langage (remarque en III.3.), qui est un type "caché".(1)
- le type mot, union des types valeur, adresse et étiquette (éti), désigne le type du contenu mémoire d'une adresse.
- le type adresse qui comprend en particulier les pointeurs MP, SP, etc ...
- le type valeur qui est tout mot ne "représentant" ni une adresse ni une étiquette. Une valeur représente un entier, un booléen, aussi bien qu'un mot d'un ensemble.
- le type étiquette, servant au branchement et aux appels et retours de procédure.

b) le "type d'intérêt" : le type modif

Il représente les instructions, ou bloc d'instructions.
Un exemple de terme de ce type : dépiler(empiler(mot 'm'))
(dont l'application à tout état S donne S!)

---

(1) Un type caché sert à la définition des autres types, mais n'apparaît pas dans la valeur sémantique d'un programme.

Quant aux principales opérations de chaque type, elles sont :

(en omettant l'opération d'égalité sur le type, de profil : (type,type) → logique)

- pour le type logique (Annexe : A.2.1.) : vrai, faux, non-log, et-log, ou-log, inf-log dont les constructeurs sont vrai et faux

- pour le type mot (A.2.1.) : une seule opération contenu qui donne le mot contenu à une adresse donnée :
  cont : (adresse) → mot, c'est le constructeur du type.

- pour le type adresse (A.2.3.) (les constructeurs sont LB et Succ-ad)
  . les pointeurs : ( ) adresse : LB, ..., SP, ..., UB
  . une opération d'indexation (adresse,valeur) → adresse : index
  . l'opération ⊕ 1 : (adresse) → adresse : succ-ad utile pour simplifier l'écriture des axiomes.
  . l'opération inf-ad : (adresse,adresse) → logique, qui donne un ordre sur les adresses.

- pour le type valeur : (A.2.5.)
  . les opérations "de bases" (constructeurs) sont valeur-min et succ (la valeur minimum et la fonction successeur), définis sur les constantes valeur '0', etc : ( ) → Valeur : valeur-min, valeur-max
    (valeur) → valeur : succ.

  . des opérations permettant de distinguer les valeurs bornées des valeurs non bornées, ainsi que les octets par rapport aux mots :
    ( ) → valeur : valeur-octet-min, valeur-octet-max
    (Valeur) → logique : mot, octet.

    mot est défini sur les valeurs non bornées, c'est-à-dire sur les valeurs sans précondition, par :
    mot(v) = (valeur-min inf-eg-valeur v) et-log(v inf-eg-valeur valeur-max)

  . des opérations permettant les calculs (définies sur les valeurs bornées soit avec la précondition mot(v)) : (valeur) → valeur : abs, non-bool...

. un ordre : (valeur,valeur) → logique : inf-val.

- Pour le type étiquette : (A.2.11)
  . une opération permettant d'étiqueter une modif :
  (eti,modif) → modif : étiqueter
  Les constructeurs sont les constantes du type (eti 'v')

- Pour le type modif : (A.2.12)
  . les opérations cachées (servant à définir les autres mais n'apparaissant pas dans la valeur sémantique d'un programme (qui est ici, rappelons-le un terme de type modif)
  empiler, dépiler, ranger (le sommet de pile à l'adresse donnée),
  empiler-mult : (adresse,valeur) → modif,
  appel-proc : (valeur,eti,adresse) → modif
  où valeur est le numéro de la procédure, eti est l'étiquette retour et a est le lien statique, qui dépend du type de l'appel (global, local, etc)

  . les opérations :
  en plus de rien et seq (séquence de modif), à chaque instruction du P-Code correspond une opération du type modif :

  exemples :
  ( ) → modif : sto(store indirect), adi(addition d'entiers), uni(union d'ensemble),... (qui prennent leurs paramètres sur la pile)

  (valeur) → modif : ldm(load multiple word)...
  (eti) → modif : ujp (saut) ...
  (valeur,eti) → modif : cpl(appel de procédure locale) ...

  . des opérations, qui doivent être vues comme des macros, sont utilisées :

  sEQ prend en compte n modifs, et se réécrit par les axiomes :

  sEQ ( ) = rien
  sEQ (m) = m
  sEQ(m1,m2) = seq(m1,m2)
  sEQ(m1,m,m2) = seq(m1,sEQ(m,m2))

. dans les définitions de modif, le si-alors-sinon est utilisé, pour rendre leur lecture plus aisée, bien qu'il ne s'agisse pas de la conditionnell habituelle :

```
appl(m,S) = si  formule (3)      ∈  S
            alors S1
            sinon S2
            finsi
```

signifie :

formule $\in S^{(1)} \Longrightarrow$ appl(m,S) = S1 et

Non formule $\in S^{(2)} \Longrightarrow$ appl(m,S) = S2.

Notons qu'il peut arriver que ni (1) ni (2) se soit vrai, (par exemple si la formule contient valeur(I) alors que I n'est pas défini).

. le si-alors-sinsi ... sinon se transforme de même :

```
appl(m,S) = si f(1) ∈ S
            alors S(1)
            sinsi ...
            sinsi f(n) ∈ S
            alors S(n)
            sinon S(n+1)
            finsi
```

= f(1) $\in$ S $\Longrightarrow$ appl(m,S) = S(1) et

...

(Non f(1) $\in$ S) et ... et (Non f(n-1) $\in$ S) et (f(n) $\in$ S)
appl(m,S) = S(n)

(Non f(1) $\in$ S) et ... et (Non f(n) $\in$ S)
appl(m,S) = S(n+1)

---

(3) rappelons qu'une formule est un axiome sans variable : valeur(I) égal 1. Nous utilisons "égal" pour distinguer cette formule (trouvée ou non trouvée dans un état), d'un axiome du type abstrait : valeur(I) = 1.

Nous donnons maintenant quelques axiomes "types" :

. Dans le type logique : (A.2.1.)

L'égalité est définie par les axiomes :

1 eg-log 1' = 1' eg-log 1, 1 eg-log 1 = vrai

vrai eg-log faux = faux où 1 et 1' sont des termes de type logique.

. Dans le type adresse : (A.2.3.)

L'opération enreg-ancêtre (utilisée dans l'opération du type modif lod[1]) de profil (valeur) → adresse est donné par les axiomes :

```
enreg-ancêtre(v) = enreg-ancêtre-1(v,MP)
enreg-ancêtre-1(v,n) = si v eg-val valeur '0'
                       alors a
                       sinon enreg-ancêtre-1(pred(v),cont(a))
```

Les préconditions sur enreg-ancêtre sont :

```
pre(enreg-ancêtre(v)) = mot-pos(v)          ("v mot positif")
pre(enreg-ancêtre-1(v,a)) = mot-pos(v)
```

Les cas d'échec sur l'indexation sont :

```
(a eg-adresse LB) et-log (v inf-val Valeur '0')
            ⟹ échec(index(a,v))
```

et de manière similaire :

```
(a eg-ad UB) et-log (v sup-val valeur '0')
        ⟹ échec(index(a,v))
```

. Dans le type valeur : (A.2.5.)

```
mot(v) = (v sup-eg-val valeur-min) et-log
         (v inf-eg-val valeur max)
```

et, ou et non sont données par l'opération test-bit, test-bit(n,v) valant valeur '1' si le bit n de v est à 1, valeur '0' sinon.

```
test-bit(n,et(v1,v2)) = si(test-bit(n,v1) eg-valeur valeur '1') et-log
                          (test-bit(n,v2) eg-valeur valeur '1')
                        alors valeur '1'
                        sinon valeur '0'
```

(1) enreg-ancêtre a été décrit dans la signification de l'instruction LOD ; enreg-ancêtre(v) est l'adresse de l'enregistrement situé à v liens statiques de l'enregistrement courant (pointé par MP)

il n'y a pas de précondition sur mot et la précondition sur test-bit est

pre(test-bit(v_1,v_2)) = (v_1 sup-eg-val valeur '0') et-log
(v_1 inf-eg-val pred(taille-mot)) et-log
mot(v_2)

où taille-mot est le nombre de bits dans un mot.

Les cas d'échec sont surtout du type :

non-logique(mot(add(v_1,v_2))) $\Longrightarrow$ échec(add(v_1,v_2))

. Dans le type modif : (A.2.12.)
(une modif m est donné par le résultat de son application à un état S :

appl(m,S1) = S2
(dans l'annexe, le signe $\in$ sera écrit Appart).

Quelques exemples :

appl(empiler(m),S) = appl(seq(subst(SP,pred-ad(SP));
subst(cont(SP),m)),S)

En utilisant les opérations cachées :

appl(sto,S) = appl(seq(ranger(cont(succ-ad(SP))),
dépiler),S)

où

appl(ranger(a),S) = appl(seq(subst(cont(a),cont(SP)),
dépiler),S)

En utilisant les opérations d'un autre type :

appl(adi,S) = appl(seq(subst(cont(succ-ad(SP)),
add(cont(succ-ad(SP)),
cont(SP))),
dépiler),S)

En utilisant la récursion :

uni est l'union de deux ensembles se trouvant sur la pile :

appl(uni,S) = appl(uni-1(cont(SP),
succ-ad(SP),
index(succ-ad-2(SP),cont(SP))),S)

où appl(uni-l(n,a1,a2),S)

= si n eg-val valeur '0'
alors appl(subst(SP,index(succ-ad(SP),
cont(SP))),S)
sinon appl(seq(subst(cont(a2),
ou(cont(a1),cont(a2))),
uni-l(pred(n),succ-ad(a1),
succ-ad(a2))),S)

l'appel de procédure : (cpl est l'appel d'une procédure locale)

appl(cpl(v,e),S) = appl(appel-proc(v,e,MP),S)

avec

appl(appel-proc(v,e,a),S) "a est le lien statique"

= appl(seq(subst(SP,index(SP),
cont(index(UB,
opp(pred(mult(v, valeur '2'))))),
"réservation de place sur la pile pour les variables de v"
subst(cont(SP),v),
"n° de procédure"
empiler(e),
"étiquette retour dans l'appelant"
empiler(MP),
"lien dynamique"
empiler(a),
"lien statique"
subst(MP,index(SP,valeur '3')),
ujp(eti 'v')),
"aller-à-la-procédure v"
S)

Les préconditions et cas d'échec de ces opérations sont :

pre(ldm(v)) = octet-pos(v)

pre(adi) = mot(cont(SP)) et-log mot(cont(succ-ad(SP)))

L'union de deux ensembles n'est licite que s'ils sont de même taille :

non-log(cont(SP)eg-valeur cont(index(SP,succ(cont(SP)))))
⟹ échec(uni)

Pour l'appel de procédure :

non-log(octet-pos(v))⟹échec(cpl(v,e))

IV.2. - Les problèmes particuliers et les principaux choix

□ Le type abstrait doit décrire complètement le langage.

Pour cela des opérations "cachées", de type modif, sont nécessaires. Elles sont ici bien séparées des autres, "vraies" opérations, qui correspondent à une instruction du langage.

Les opérations cachées ne servent qu'à définir les "vraies" opérations. Elles n'apparaissent jamais dans la valeur sémantique d'un programme.

□ Deux cas d'échec distincts sont utilisés, l'un est l'échec "normal", l'autre l'échec optionnel : ces échecs ont été définis en III.3)

. le premier est le cas d'échec d'une opération dont les paramètres n'ont pas la forme voulue.
Exemple : v eg-val valeur '0' ⟹ échec(div(v,v1)
ou non-log(cont(SP) eg-val
cont(index(SP,succ(cont(SP)))))
⟹échec(uni) "échec de l'union de deux ensembles"

. le deuxième est, ici, le cas de débordement de pile
Il s'agit d'un échec optionnel, c'est-à-dire que l'on connaît les raisons de cet échec lorsqu'il arrive mais on ne peut pas dire que tel état implique systématiquement l'échec.
Cette méthode de spécification laisse la possibilité à l'implémenteur du type abstrait de prévoir, par exemple, un ramasse-miette afin d'éviter l'arrêt en erreur sur débordement de pile.

On notera cet échec échec-deb, deb pouvant être un paramètre :
Exemple : échec-déb(empiler(m)) $\Longrightarrow$ SP eg-ad succ-ad(NP)

□ Le type abstrait manipule des mots et des octets, les deux étant du type mot. Mais les opérations du type modif correspondant à des instructions qui demandent des octets sont munies de préconditions ou cas d'échec du type :
pre(f(x)) = octet(x).

□ Pour les ensembles, représentés par des blocs de mot, nous utilisons des opérations travaillant sur des entiers (valeurs) à l'aide de puissance et modulo.
Mais nous ne pouvons nous affranchir du choix de représentation machine de ces entiers ; trois opérations utilisent le fait que les entiers sont complémentés à deux : test-bit, inv-bit-de-signe et fixe-bit.

□ Les entiers bornés peuvent être définis commodément à partir des entiers non bornés, donc :

- le type valeur comporte des opérations définies sur "les valeurs non bornées" (soit sans la précondition mot(v)) et des opérations définies sur "les valeurs bornées" (soit avec précondition) : (le type valeur a été introduit en IV.2)
  zéro, succ-valeur pred-valeur, opp-valeur, add-valeur, mult-valeur, inf-eg-valeur et mot sont définis sans précondition.
  Toutes les autres opérations du type valeur sont définies avec la précondition mot(v) : succ,valeur-min,add,...,inf-eg,...

- les constructeurs du type valeur (bornées) sont valeur-min et succ. Mais il faut définir ces opérations, ainsi que les constantes (Valeur '31' pouvant apparaître dans un programme.
  . valeur-min,succ et valeur-max sont définis par :
  succ(v) = succ-valeur(v)
  la précondition mot(v) sur succ(v) interdisant cette équation en dehors de valeur-min et valeur-max.
  Valeur-min et valeur-max sont définis selon la machine cible, à l'aide de zéro, succ-val, mult-val...

Les constantes entières sont définies grâce aux valeurs non bornées et au types text :

```
Valeur '0' = zéro, ..., valeur '9' = succ-val(valeur '8')
Valeur 'nat.c' = add-valeur(mult-valeur(succ-valeur(valeur '9'),
                                        valeur 'nat'),
                            valeur 'c')
Valeur '-nat' = opp-valeur(valeur 'nat).
```

- Les réels n'ont jamais été axiomatisés et ne peuvent l'être dans le cadre choisi.

- L'état initial de la mémoire, avant l'exécution d'un programme P-Code, dépend de ce programme.

Les zones mémoires concernées sont la zone de constante, la zone des attributs des procédures et l'enregistrement (variables locales et marque) du programme principal.

Une solution au niveau du type abstrait est de considérer que l'état initial associé au type objet prendra en compte les propriétés indépendantes du programme, ainsi que les propriétés qui en dépendent.

L'état initial comporte donc les propriétés usuelles, qui sont les formules dérivées des axiomes du types abstrait (axiomes spécifiques au langages) soit

add(x,y) = ..., MP = index(SP,valeur '3'), etc...

Mais il comporte en plus les formules concernant NIL, CP, BP, SP et NP, ainsi que les zones de constantes, d'attributs des procédures et le premier enregistrement.

Dans le cas d'une traduction langage source → P-Code, ces dernières formules seront construites lors de la traduction car elles dépendent en fait du programme source.

□ Le type étiquette et les modifications qui l'utilisent méritent une attention particulière.

étiqueter est l'opération principale du type eti ;

type eti

Opérations :

(eti,modif) → modif : étiqueter
(modif,eti) → logique : entrée, sortie
(eti,eti) → logique : eg-eti

avec :

entrée(m,e) est vraie si m comporte une modif étiqueter par e
sortie(m,e) est vraie si m comporte un saut à l'étiquette e
avec entrée(m,e) eg-logique faux.

entrée ne sert qu'à la définition de sortie
sortie sert à définir la séquence, les branchements et les procédures.

Remarque : Les termes du type abstrait doivent être corrects pour les étiquettes en ce sens qu'il n'y a qu'une modif étiquetée par une étiquette donnée et qu'à chaque saut correspond une étiquette. Ceci est vérifié à l'élaboration d'un terme. De plus, nous ne traitons que certains termes, ceux qui peuvent représenter un terme du type source. Ceci, bien qu'a priori génant pour l'esprit, ne l'est pas vraiment dans ce cas car "le P-Code est un langage dans lequel ne peuvent être écrits que des programmes corrects".

Le type étiquette ne comporte qu'un axiome, celui qui donne l'égalité sur le type par l'égalité syntaxique (puisque ce type ne comporte que des constantes.

Les axiomes donnant entrée et sortie pour chaque modif sont donnés dans le type modif :

exemples : entrée(étiqueter(e1,m),e2) = si e1 eg-eti e2
alors vrai
sinon entrée(m,e2)
sortie(étiqueter(e1,m),e2) = si e1 eg-eti e2
alors faux
sinon sortie(m,e2)
entrée(seq(m1,m2),e) = entrée(m1,e) ou-log entrée(m2,e)
sortie(seq(m1,m2),e) = si entrée(m1,e) ou-log
entrée(m2,e)
alors faux
sinon sortie(m1,e) ou-log
sortie(m2,e)
"ujp est le saut inconditionnel" :
entrée(ujp(e1),e2) = faux
sortie(ujp(e1),e2) = e1 eg-eti e2
"de même pour fjp (sortie si cont(SP) = valeur '1'(vrai))"
"cpl est l'appel de procédure locale"
entrée(cpl(v,e1),e2) = faux
sortie(cpl(v,e1),e2) = e2 eg-eti eti 'v'

Pour la plupart des autres modif m (accès en mémoire, calcul sur la pile etc ...)
entrée(m,e) = faux = sortie(m,e)

La définition de modif utilisant sortie sont par exemple :

. pour la séquence de modif, et la modif étiquetée :

sortie(m,e) = faux $\Longrightarrow$ appl(étiqueter(e,m),S)
= appl(m,S)
appl(seq(étiqueter(e,m1),m2),S)
appl(étiqueter(e,seq(m1,m2)),S)

Pour tout e sortie(m,e) = faux $\Longrightarrow$ appl(seq(m1,m2),S)
= appl(m2,appl(m1,S))

(si il existe une étiquette e telle que sortie(m,e) = vrai
l'axiome en partie droite est donc peut-être faux)

. Pour les sauts en avant :

appl(sEQ(ujp(e),m1,étiqueter(e,m2)),S)
= appl(étiqueter(e,m2),S)

. Pour les boucles :

Soit m = étiqueter(boucle,sEQ(m1,tjp(ext),m2,ujp(boucle),
étiqueter(ext,m3)))

avec pour toute sortie(m1,e) = faux
et sortie(m2,e) = faux

cont(SP) = valeur '1' $\in$ appl(m1,S)
appl(m,S) = appl(sEQ(m1,dépiler,étiqueter(ext,m3)),S)

cont(SP) = valeur '0' $\in$ appl(m1,S)
appl(m,S) = appl(m,appl(sEQ(m1,dépiler,m2),S))

. Pour les procédures

On suppose que pour toute sortie(m1,e) = faux
et que e eg-eti e3 $\in$ appl(m1,S) ;

appl(sEQ(ujp(e1,étiqueter(e2,m2),ujp(e),
étiqueter(e1,m1),ujp(e2),
étiqueter(e3,m3)),S)
= appl(sEQ(ujp(e1),étiqueter(e2,m2),ujp(e),
étiqueter(e1,m1),m2,
étiqueter(e3,m3)),S)

où la première modif représente le programme :

déclaration de la procédure e2 (avec son instruction de retour ujp(e),
un bloc d'instruction m1, un appel à e2 et un bloc d'instruction e3.

On remarque que dans le cas des boucles comme dans celui des appels et retour de procédures, il y a duplication d'une modif m lors de l'évaluation de

appl(sEQ(m1,m,m2),S)

Ceci introduit un problème : le dupliquage éventuel des étiquettes apparaissant dans m.

La solution est un renommage des étiquettes, qui sera dynamique car le nombre d'appels de procédure, comme le nombre de tours dans la boucle, ne peut être déterminé statiquement.

Notons que ce renommage n'intervient pas dans Perluette mais dans la fonction appl(m,S), où m est le terme objet et S l'état initial objet qui donne la sémantique de ce terme.

- Les instructions P-Code correspondant à l'appel de procédure n'ont pas parmi leurs paramètres l'adresse retour dans l'appelant. Cette adresse est fournie par le compteur ordinal qui n'est pas une donnée du langage mais de l'interpréteur de ce langage.

Dans le type modif, nous avons considéré, pour les branchements des étiquettes et non des adresses précises dans le code.

Ainsi nous donnons une étiquette (l'étiquette retour) en paramètre de l'appel de procédure. Cette étiquette ne peut être fournie que par le compilateur (soit lors de la représentation d'un terme source par un terme objet).

Ce choix n'est pas éloigné du monde réel puisque certains compilateurs fournissent en effet cette étiquette retour à l'appel de procédure.
Il permet de plus une spécification claire et cohérente que n'aurait pas pu permettre l'introduction brutale d'un compteur ordinal.

V. - CONCLUSION

Les types abstraits s'avèrent un outil relativement puissant et clair pour donner la sémantique d'un langage de programmation. Mais l'expérience montre la nécessité d'un langage de spécification possédant les types paramétrés et des "schémas de formules".

L'originalité de ce travail est la donnée d'une sémantique d'un langage de bas niveau.
Jusqu'ici, on n'a donné la sémantique que de langages de haut niveau, plus complexes et aussi plus beaux.
Pourtant le manuel du P-Code a les mêmes lacunes que le manuel du Pascal : il ne peut résoudre toutes les ambiguïtés, il est difficile à lire, ...
Dans le cas d'un langage de bas niveau puissant, il induit de plus en erreur, en parlant de pile, d'ensemble, de procédure,... qui n'ont pas de réalité.
Or dans une méthode où l'on spécifie un traducteur d'un langage source vers un langage objet, sans fixer a priori le second, il faut donner la sémantique des deux langages. Et le deuxième peut-être de bas niveau.

A - ANNEXES

A.1. - Différences entres le P-Code UCSD (version IV.0) et celui choisi

Les principales différences sont la suppression des instructions qui n'ajoutent rien à la puissance du P-Code, la clarification de langage et de la mémoire, les modifications dues au choix du Pascal ISO au lieu du Pascal UCSD.

Il n'y a donc rien de fondamental, mais la liste exhausive est longue. (Une connaissance de P-Code UCSD est nécessaire pour la lecture de ce qui suit)

- le P-Code est un langage d'assemblage et non un langage machine : le code est en mémoire, mais à part, dans une région non adressable par lui-même (ainsi il ne peut se modifier !), les sauts se font à des étiquettes et non à une adresse. Le langage gagne ainsi en clareté et en portabilité.

- l'organisation de la mémoire est simplifiée, pour clarifier son dessin, et la spécification des pointeurs vers des zones mémoires données.

- Il n'y a qu'une seule unité de compilation (d'ou un seul "segment"). Les instructions modifiées sont LDE,LAE,CSP et MOV.
  Les appels de procédures externes sont supprimés. MSENV est oté de la marque.

- Les pointeurs MP, NP, BP sont des adresses particulières considérés comme des registres.

- Les réels occupent deux mots ; la taille des réels étant un paramètre caduque dès que les opérations sur ceux-ci sont cablées.

- Le genre de l'octet était un paramètre permettant de s'affranchir du sens de "lecture" d'un mot (le bit faible pouvant être à gauche ou à droite). Ceci imposait des précautions lors des calculs. Ce paramètre est supprimé.

- Les instructions faites pour manipuler des octets ou des champs de bits sont supprimées. Elles étaient là pour tenir compte efficacement des structures "compactifiées" ("packed array"). Mais ceci n'était pas fondamental.

- Les instructions "compacte" rajoutées dans la version IV par rapport à la version II sont supprimées. Exemple d'instruction compacte : SLDC5, sans argument qui est équivalent à un LDCI avec l'argument 5.

- Les instructions manipulant des chaînes, faites pour traduire le Pascal UCSD et non le Pascal ISO, sont supprimées.

- L'instruction CJP, correspondant au case du Pascal UCSD et non à celui du Pascal ISO, est modifiée.

- Petites modifications dans les comparaisons de structures (les instructions ajoutées figuraient dans la version II) :

  LEUSW et GEUSW, comparaison d'entiers non signés, sont supprimées
  LESI, GTRI, NEQR, LESR, GTRR, NEQS, comparaison d'entiers, de réels ou d'ensembles, sont ajoutés.
  LDCRL, LDRL, STRL inutiles si les réels occupent deux mots, sont supprimées. (les noms des instructions ont parfois été changés pour plus d'homogénéité).

- Les appels des procédures ne fonctionnaient pas uniquement avec les données du P-Code, mais sans doute grâce à des tables de son interpréteur (qui contenaient sans doute, entre autres, les zones de marque) :

La zone de marque est mise en avant la pile d'évaluation. Le niveau lexical est rajouté dans les attributs des procédures. Dans les marques figure le numéro de la procédure et non celui de l'appelant.

- CPP : appel de procédure prédéfinie (write par exemple) est ajoutée. Ce cas est en effet particulier car on sait seulement que le résultat éventuel de la procédure sera en sommet de pile, mais on ne connaît pas le corps de la procédure.

- Le code n'étant plus en mémoire, LCO qui donnait un déplacement par rapport à la base de la zone de code est changé en LAC qui fournit une adresse.

- La sortie d'une procédure (EXIT P dans la procédure Q faisait retourner après le premier appel de P dans la chaîne dynamique des appels) peut être incorrecte sans que l'on puisse s'en apercevoir statiquement. Comme l'on peut toujours (en introduisant des variables globales) modifier le programme source de manière à sortir des procédures normalement, EXIT est supprimé.

- Les instructions "Systèmes" et non "P-Code" (en cas de code natif par exemple) sont supprimées : LPR, SPR, NAT, NAT-INFO.

A.2. - Le type abstrait P-Code

P_code.

```
        %           Un type abstrait          %
        %        decrivant un P-Code          %

%
Le P-Code ne comporte que des instructions donc :

     Le type modif a presque toutes ses operations visibles,
     tandis que le type logique est cache, et que le type mot
     (adresse, valeur, eti) a quelques operations visibles.
%

Type logique : %type cache%
%************%


   Operations :
   %**********%

     ()                    -> logique    : vrai, faux ;
     (logique)             -> logique    : non-log ;
     (logique,logique)     -> logique    : et-log, ou-log, eg-log,
                                           inf-log, sup-log, inf-eg-log,
                                           sup-eg-log (infixe) ;

   Axiomes :
   %*******%
     non-log (vrai)     == faux ;
     non-log (faux)     == vrai ;
     l ou-log l'        == si l eg-log vrai
                           alors vrai
                           sinon l'
                           finsi ;
     l et-log l'        == non-log((non-log(l)) ou-log(non-log(l'))) ;
     l eg-log l'        == l' eg-log l ;
     l eg-log l         == vrai ;
     vrai eg-log faux   == faux ;
     l inf-log l'       == (l eg-log faux) et-log (l' eg-log vrai) ;
     l sup-log l'       == l' inf-log l ;
     l inf-eg-log l'    == (l inf-log l') ou-log (l eg-log l') ;
     l sup-eg-log l'    == l' inf-eg-log l ;

fin logique ;
%***********%
```

```
Type mot = union (valeur, adresse, eti) :
%****************************************%


   Operations : %cachees%
   %**********%

     (adresse) -> mot : cont ;


   cas d'echec :
   %***********%

     echec(cont(a)) <= (a sup-ad NP) et-log (a inf-ad SP) ;

fin mot ;
%*******%



Type adresse :
%************%


   Operations :
   %**********%

     ()                  -> adresse : LB, NP, CP, NIL, UB ;
     ()                  -> adresse : SP, MP, BP (mod) ;
     (adresse)           -> adresse : succ-ad ;

   %Operations cachees :%
   %********************%

     (adresse)            -> adresse : succ-ad-2,
                                       pred-ad ;
     (adresse, valeur) -> adresse : index ;
     (valeur)          -> adresse : enreg-ancetre ;
     (valeur, adresse) -> adresse : enreg-ancetre-1 ;
     (adresse, adresse)-> logique : eg-ad, inf-ad, sup-ad ;

   Axiomes :
   %*******%
     pred-ad(a)     == index(a,opp(valeur '1')) ;
     succ-add-2(a)  == succ-ad(succ-ad(a)) ;
     index(a,v)     == si v eg-val valeur '0'
                       alors a
                       sinon si v sup-val valeur '0'
                             alors succ-ad(index(a,
                                                 pred(v)))
                             sinon pred-ad(index(a,
                                                 succ(v)))
```

```
                        finsi
                finsi ;


    enreg-ancetre(v)          == enreg-ancetre-1 (v, MP) ;
    enreg-ancetre-1 (v,a)     == si v eg-ad valeur '0'
                                 alors a
                                 sinon enreg-ancetre-1(pred(v),
                                                       cont(a))
                                 finsi ;

    LB eg-ad LB                  == vrai ;
    LB eg-ad succ-ad(a)          == faux ;
    succ-ad(a) eg-ad LB          == faux ;
    succ-ad(a) eg-ad succ-ad(a') == a eg-ad a' ;

    a inf-ad a' == si a eg-ad a'
                   alors faux
                   sinon si a eg-ad LB
                         alors vrai
                         sinon pred-ad(a) inf-ad a'
                         finsi
                   finsi ;
    a sup-ad a'   == a' inf-ad a ;


  Preconditions :
  %*************%


    pre(enreg-ancetre(v))     == mot-pos(v) ;
    pre(enreg-ancetre-1(v,a))== mot-pos(v) ;

  Cas d'echec :
  %***********%

    (a eg-ad LB) et-log (v inf-val valeur '0')   => echec(index(a,v));
    (a eg-ad UB) et-log (v sup-val valeur '0')   => echec(index(a,v)) ;
    a eg-ad LB                                   => echec(pred-ad(a)) ;
    a eg-ad UB                                   => echec(succ-ad(a)) ;
    a eg-ad (pred-ad (UB))                       => echec(succ-ad-2(a)) ;

fin adresse ;
%*********%


Type valeur :
%***********%

Operations :
%**********%
```

```
( )          -> valeur : valeur-min, valeur-max ;
(valeur)     -> valeur : succ;

%Operations cachees :%
%**********************%

   %pour les acces, les adresses, les entiers,et les booleens:%

      ( )                -> valeur :valeur-octet-min, valeur-octet-max ;
      (valeur)           -> valeur  : pred, opp, abs, non-bool,
                                      exp-2 ;
      (valeur,valeur)    -> valeur  : add, sous,  mult, div-ent, mod,
                                      eg, non-eg, inf, inf-eg, sup,
                                      sup-eg ;
      (valeur,valeur)    -> logique : eg-val, inf-val, sup-val,
                                      inf-eg-val, sup-eg-val (infixe) ;

%
 operations sur les "valeurs entieres" non bornees, c'est-a-dire
 sans precondition :
%

      (valeur)           -> valeur   : zero, pred-valeur, succ-valeur,
                                       cpp-valeur ;
      (valeur,valeur)    -> valeur   : add-valeur, mult-valeur ;
      (valeur,valeur)    -> logique  : inf-eg-valeur ;
      (valeur)           -> logique  : mot, octet, mot-pos, octet-pos ;


%pour les ensembles :%

      ()                 -> valeur : taille-mot   %nb de bits dans 1 mot%,
                                     card-1       %cardinal max d'un ens-1% ;
      (valeur)           -> valeur : non, inv-bit-de-signe ;
      (valeur,valeur)    -> valeur : ou, et, test-bit, fixe-bit ;
      (valeur,valeur,valeur)
                         -> valeur : incl-sur-n-bits ;

%les procedures :%

   (valeur,adresse) -> adresse : lien-stat ;

%
lien-stat(v,a) remonte les liens statiques, a partir de l'enre-
gistrement d'adresse a, v fois et rend le lien stat. de l'enregistrement
trouve
%

   axiomes :
   %*******%

%pour les adresses, acces, entiers et booleens%

      succ(v)                          == succ-valeur(v) ;
```

```
pred(succ(v))                  == v ;
opp(succ(valeur-min))          == valeur-max ;
opp(succ(valeur-octet-min)) == valeur-octet-max ;
opp(valeur-max)                == succ(valeur-min) ;
opp(succ(succ(pred(v))))       == pred(opp(pred(succ(v)))) ;

add(v,v')      ==   si v eg-val valeur '0'
                    alors v'
                    sinon si v inf-val valeur '0'
                          alors si v' inf-val valeur '0'
                                alors opp(add(opp(v),opp(v')))
                                sinon add(v',v)
                                finsi
                          sinon succ(add(pred(v),v'))
                          finsi
                    finsi ;


abs(v)       == si v sup-eg-val valeur '0'
                alors v
                sinon opp(v)
                finsi ;

non-bool(v) == si v eg-val valeur '0'
                alors valeur '1'
                sinon valeur '0'
                finsi ;


exp-2(v)     == si v eg-val valeur '0'
                alors valeur '1'
                sinsi v inf-val valeur '0'
                alors mult(valeur '2', exp-2(pred(v))
                sinon div(valeur '1',
                          exp-2(opp(v)))
                finsi ;

sous(v,v1) == si v1 eg-val valeur '0'
              alors v
              sinon si v1 inf-val valeur '0'
                    alors add(v,opp(v1))
                    sinon pred(sous(v,pred(v1)))
                    finsi
              finsi ;

mult(v,v1)  == si v1 eg-val valeur '0'
               alors valeur '0'
               sinon si v1 inf-val valeur '0'
                     alors si v inf-val valeur '0'
                           alors mult(opp(v),opp(v1))
                           sinon opp(mult(v,v1))
                           finsi
                     sinon add(mult(v,pred(v1)),v)
                     finsi
                finsi ;
```

```
div-ent(v,v1) == si (v sup-eg-val valeur '0') et-log
                    (v1 sup-val valeur '0')
                 alors si v inf-val v1
                       alors valeur '0'
                       sinon succ (div-ent(sous(v,v1),v1))
                       finsi
                 sinon si (v inf-val valeur '0') et-log

                          (v1 inf-val valeur '0')
                       alors div-ent(opp(v),opp(v1))
                       sinon  si (v inf-val valeur '0') et-log
                                 (v1 sup-val valeur '0')
                              alors opp(div-ent(opp(v),v1))
                              sinon opp(div-ent(v,opp(v1)))
                              finsi
                        finsi
                  finsi ;


mod(v,v1)    ==  si v sup-eg-val valeur '0'
                 alors si v inf v1
                       alors v
                       sinon mod(sous(v,v1),v1)
                       finsi
                 sinon mod(add(v,v1),v1)
                 finsi ;
```

%axiomes sur les valeurs non bornees :%

```
succ-valeur(pred-valeur(v))    == v ;
pred-valeur(succ-valeur(v))    == v ;
opp-valeur(zero)               == zero ;
opp-valeur(pred-valeur(v))     == succ-valeur(opp-valeur(v)) ;
opp-valeur(succ-valeur(v))     == pred-valeur(opp-valeur(v)) ;
add-valeur(v,zero)             == v ;
add-valeur(v,succ-valeur(v'))  == succ-valeur(add-valeur(v,v1)) ;
add-valeur(v,pred-valeur(v1))  == pred-valeur(add-valeur(v,v1)) ;
mult-valeur(v,zero)            == zero ;
mult-valeur(v,succ-valeur(v1))== add-valeur(mult-valeur(v,v1),
                                            succ(valeur(v1))) ;
mult-valeur(v,pred-valeur(v1))== add-valeur(mult-valeur(v,v1),
                                              pred-valeur(v1)) ;
mult-valeur(v,v1)              == mult-valeur(v1,v) ;
inf-eg-valeur(zero,zero)       == vrai ;


inf-eg-valeur(zero,pred-valeur(zero))          == faux ;
inf-eg-valeur(succ-valeur(v), succ-valeur(v1)) == inf-eg-valeur
                                                  (v,v1) ;

inf-eg-valeur(zero,succ-valeur(v))
                               == si inf-eg-valeur(zero,v)
                                  alors vrai
```

```
                                    sinon inf-eg-valeur
                                          (zero,succ-valeur(v))
                                    finsi ;


    inf-eg-valeur(zero,pred-valeur(v))
                                 == si inf-eg-valeur(zero,v)
                                    alors inf-eg-valeur
                                          (zero,pred-valeur(v))
                                    sinon faux
                                    finsi ;

%l'egalite sur le type valeur :%

    valeur-min eg-val valeur-min == vrai ;
    valeur-min eg-val succ(v)    == faux ;
    succ(v) eg-val succ(v')      == v eg-val v' ;
    succ(v) eg-val valeur-min    == faux ;

%definition d'un ordre sur les valeurs :%

    v inf-val v'     == si v eg-val v'
                        alors faux
                        sinon si v      eg-val valeur-min
                              alors vrai
                              sinon pred(v) inf-val v'
                              finsi
                        finsi ;
    v sup-val v'     == v' inf-val v ;
    v sup-eg-val     == (v sup-val v') ou-log (v eg-val v') ;
    v inf-eg-val     == v' sup-eg-val v ;

%
les operations de comparaison rendant des "booleens" du
langage :
%

    eg(v,v')    == si v eg-val v'
                   alors valeur '1'
                   sinon valeur '0'
                   finsi ;

    non-eg(v,v')
                == non-bool(eg(v,v')) ;
    inf-eg(v,v')
                == si v inf-eg-val v'
                   alors valeur '1'
                   sinon valeur '0'
                   finsi ;

    sup-eg(v,v')
                == inf-eg(v',v) ;
```

```
      inf(v,v')  == et(inf-eg(v,v'), non-eg(v,v')) ;
      sup(v,v1)  == inf(v',v) ;

%pour les preconditions et cas d'echec :%

      mot(v)      == (inf-eg-valeur(valeur-min,v)) et-log
                     (inf-eg-valeur(v,valeur-max)) ;

      octet(v)    == (inf-eg-valeur(valeur-octet-min,v)) et-log
                     (inf-eg-valeur(v,valeur-octet-max)) ;

      mot-pos(v) == (mot(v) et-log (inf-eg-valeur(valeur '0' ,v)) ;

      octet-pos(v)
                  == octet(v) et-log (inf-eg-valeur(valeur '0',v)) ;

%pour les ensembles :%

%taille-mot = log-base-2(valeur-max + valeur '1')%


%test-bit(n,v) est 'vrai' si le bit n de v est a 1 :%

      test-bit(n,v)    == si n eg-val pred (taille-mot)
                          alors inf-eg(v,valeur '0')
                          sinon eg(div-ent(exp-2(n)),mod(v,exp-2(succ(n))),
                                   valeur '1')
                          finsi ;

%et, ou et non sont donne par test-bit :%

      test-bit(n,et(v1,v2)) == si test-bit(n,v1) eg-val valeur '1'
                                                 et-log
                               test-bit(n,v2) eg-val valeur '1'


                               alors valeur '1'
                               sinon valeur '0'
                               finsi ;

      test-bit(n,ou(v,v2))   == si test-bit(n,v1) eg-val valeur '0'
                                                  et-log
                                test-bit(n,v2) eg-val valeur '0'
                               alors valeur '0'
                               sinon valeur '1'
                               finsi ;

      test-bit(n,non(v))     == si test-bit(n,v) eg-val valeur '1'
                               alors valeur '0'
                               sinon valeur '1'
                               finsi ;


      inv-bit-de-signe(v)    == si v sup-eg-val valeur '0'
                               alors add(v,valeur-min)
                               sinon opp(add(opp(v), valeur-min))
                               finsi ;
```

```
incl-sur-n-bits(n,v,v2)
                        == si n eg-val valeur '0'
                           alors valeur '1'
                           sinon  si test-bit(n,v1) eg-val
                                                valeur '1' et-log
                                     test-bit(n,v2) eg-val
                                                valeur '0'
                                  alors valeur '0'
                                  sinon incl-sur-n-bits(pred(n),
                                                        v1,v2)
                                  finsi
                            finsi ;

fixe-bit(n,m) == si m eg-val pred(taille-mot)
                 alors inv-bit-de-signe(fixe-bit(n,pred(m)))
                 sinsi si m eg-val pred(pred(taille-mot))
                 alors mult(div-ent(valeur-max, exp-2(n)),
                            exp-2(n))
                 sinon mod(mult(div-ent(valeur-max,exp-2(n)),
                                        exp-2(n)),
                           exp-2(succ(m))
                 finsi ;

%les procedures :%

lien-stat(v,a) == si v eg-val valeur '0' alors cont(a)
                  sinon lien-stat(pred(v),cont(a))
                  finsi ;


Preconditions :
%**************%

%pour les acces, les adresses, les entiers et les booleens%

pre(succ(v))  == mot(v) et-log non-log(v eg-val valeur-max) ;
pre(pred(v))  == mot(v) et-log non-log(v eg-val valeur-min) ;
pre(opp(v))   == mot(v) et-log non-log(v eg-val valeur-min) ;
pre(abs(v))   == mot(v) et-log non-log(v eg-val valeur-min) ;
pre(exp-2(v)) == mot(v) ;

pre(add(v,v'))
              == mot(v) et-log mot(v') ;

%de meme pour sous, mult, eg, non-eg, inf, inf-eg, sup, sup-eg%

pre(non-bool(v)) == (v eg-val valeur '0') ou-log
                    (v eg-val valeur '1') ;
pre(mod(v,v'))   == (v' sup-val valeur '0') et-log mot(v)
                                            et-log mot(v') ;
pre(div-ent(v,v'))
                 == non-log(v' eg-val valeur '0') et-log mot(v)
```

```
                                                    et-log mot(v') ;

%pour les ensembles :%

     pre(non(v))                          == mot(v) ;
     pre(inv-bit-de-signe(v))             == mot(v) ;
     pre(ou(v1,v2))                       == mot(v1) et-log mot(v2) ;
     pre(et(v1,v2))                       == mot(v1) et-log mot(v2) ;
     pre(test-bit(v1,v2))                 == (v1 sup-eg-val valeur '0') et-log
                                             (v1 inf-eg-val pred(taille-mot)) et-log
                                             mot(v2) ;
     pre(fixe-bit(v1,v2))                 == (v1 sup-eg-val valeur '0') et-log
                                             (v1 inf-eg-val pred(taille-mot)) et-log
                                             (v2 sup-eg-val valeur '0') et-log
                                             (v2 inf-eg-val pred(taille-mot)) ;

     pre(incl-sur-n-bits(v1,v2,v3))
                                          == (v1 sup-eg-val valeur '0') et-log
                                             (v1 inf-eg-val pred(taille-mot)) et-log
                                             mot(v2) et-log mot(v3) ;

     pre(lien-stat(u,a))                  == mot-pos(v) ;

   cas d'echec :
   %**********%

%pour les acces, les adresses, les entiers et les booleens :%

     non-log (mot(add(v,v'))) => echec(add(v,v')) ;

%de meme pour sous, mult%

     non-log mot(div-ent(v,v')) ou-log (v' eg-val valeur '0')
                                => echec(div-ent(v,v')) ;
     v' inf-eg-val valeur '0' => echec(mod(v,v')) ;
fin valeur ;
%**********%

%

Type eti :
%*********%

   operations :
   %**********%

     (eti,modif) -> modif   : etiqueter ;
     (modif,eti) -> logique : entree, sortie ;
     (eti,eti)   -> logique : eg-eti ;

%
entree(m,e) est vraie si m comporte une modif etiquetee par e
sortie(m,e) est vraie si m comporte un saut a l'etiquette e
                    avec entree(m,e) = faux
```

```
entree ne sert qu'a definir sortie dans la sequence de modif
sortie sert a definir la sequence, les boucles et les procedures
%

%
les constructeurs du type etiquette etant des constantes, eg-eti est
l'egalite syntaxique :
%

   axiomes :
   %********%


     eti 'e' eg-eti eti 'e' ==e egs e' ;


%entree et sortie d'une modif :%

     entree(etiqueter(e,m),e')  == si e eg-eti e' alors vrai
                                   sinon entree(m,e')
                                   finsi ;
     sortie(etiqueter(e,m),e')  == si e eg-eti e' alors faux
                                   sinon sortie(m,e')
                                   finsi ;
     entree(seq(m,m'),e)        == entree(m,e) ou-log entree(m',e)
     sortie(seq(m,m'),e)        == si entree(m,e) ou-log entree(m',e)
                                             alors faux
                                   sinon sortie(m,e) ou-log sortie (m',e)
                                   finsi ;


     entree(ujp(e),e')          == faux ;
     sortie(ujp(e),e')          == e eg-eti e' ;
     entree(fjp(e),e')          == faux ;

%de meme pour tjp, negj et egj%

     sortie(fjp(e),e')          == e eg-eti e' ;

%de meme pour tjp, negj et egj%

     entree(cjp(v),e)           == faux ;
     sortie(cjp(v),e)           == vrai ;

     entree(cpl(v,e),e')        == faux ;
     sortie(cpl(v,e),e')        == e' eg-eti 'v' ;

%de meme pour cpg, cpp, appel-proc(v,e,a)%

     entree(cpi(v1,v2,e),e')    == faux ;
     sortie(epi(v1,v2,e),e')    == vrai ;
     entree(rpu(v))             == faux ;
     sortie(rpu(v))             == vrai ;
```

```
%pour les modif m non citees precedement :%

      entree(m,e)                    == faux ;
      sortie(m,e)                    == faux ;
fin eti ;
%********%


Type modif :
%***********%

    operations %cachees% :
    %********************%


%
ne servent qu'a la definition des operations du type modif
elles n'apparaissent jamais dans un terme objet :
%

%pour les acces, les adresses, les entiers, les booleens :%

      ()          -> modif  :  depiler, depiler-2 ;
      (mot)       -> modif  :  empiler ;
      (adresse)   -> modif  :  ranger ;

      %ranger le sommet de pile a l'adresse indiquee%

      (adresse,valeur) -> modif : empiler-mult, ldm1,stm1,stm2 ;
      (adresse,valeur,adresse)
                       -> modif : mov1, mov2 ;


%pour les ensembles :%

      (valeur)                                  -> modif : adj-1, adj-2 , inn-1 ;
      (valeur,adresse,adresse)                  -> modif : uni1, int1 ;
      (valeur,valeur)                           -> modif : srs-1 ;
      (valeur,valeur,valeur)                    -> modif : srs-2 ;
      (adresse,adresse,valeur,adresse) -> modif : equs-1, neqs-1,leqs-1 ;


%pour les procedures :%

      (valeur,eti,adresse)                      -> modif : appel-proc ;
      (valeur,eti)                              -> modif : rpu-1 ;


%operations :%
%************%

%pour les acces, les adresses, les entiers et les booleens :%

      ()              -> modif : rien,sldc0, sldc31, ldcn, sldl1,..sldl16,
                                 sldo1..sldo16,  sind0..sind7,
                                 sto, bnot, abi, ngi, inci, deci, adi,
                                 sbi, mpi, dvi, modi, chk, equi, neqi,
```

```
                                legi, lesi, geqi, gtri ;

      (valeur)        -> modif : ldcb, ldci, lac, ldl, lla, stl, ldo,
                                 lao, sro, ind, ldm, stm, mov, inc, ixa ;

      (valeur,valeur) -> modif : lod, lda, str, ldc ;
      (modif,modif)   -> modif : seq ;

 %pour les ensembles :%

      ()              -> modif : inn, uni, int, dif, equs, neqs, leqs, geqs ,
                                 srs ;

      (valeur) -> modif : adj ;

 %pour les branchements :%

      (eti)     -> modif : ujp, fjp, tjp, negj, egj ;
      (valeur) -> modif : cjp ;

 %les procedures :%

      (valeur,eti)          -> modif : cpl, cpg, cpp ;

                                    %
                                    valeur est le numero de la procedure
                                    appelee eti est l'etiquette de retour
                                    %

      (valeur,valeur,eti) -> modif : cpi ;
      (valeur)            -> modif : rpu ;



    definitions :
    %***********%


 %pour les acces, les adresses, les entiers, les booleens :%

 %les operations cachees :%

      appl(depiler,S)  == appl(subst(SP,succ-ad(SP)),S) ;
      appl(depiler-2,S)== appl(subst(SP,succ-ad-2(SP)),S) ;
      appl(empiler(m),S)
                       == appl(seq(subst(SP,pred-ad(SP)),
                                   subst(cont(SP),m)),S) ;
      appl(ranger(a),S)
                       == appl(seq(subst(cont(a),cont(SP)),
                                   depiler),S) ;
      appl(empiler-mult(a,v),S)
                       == si v egal valeur '0'
                          alors S
                          sinon appl(seq(empiler(cont(a)),
```

```
                                        empiler-mult(pred-ad(a),
                                                     pred(v))),S)
                        finsi ;

    appl(mov1(a1,v,a2),S)
                    == appl(seq(depiler-2,mov-2(a1,v,a2)),S) ;

    appl(ldm1(a,v),S == appl(seq(depiler,empiler-mult(a,v)),S) ;

    appl(stm1(a,v),S)
                    == appl(seq(stm2(a,v),depiler),S) ;

    appl(stm2(a,v),S)
                    == si v egal valeur '0'
                       alors S
                       sinon appl(seq(ranger(a),stm2(succ-ad(a),
                                                     pred(v))),S)
    appl(mov-1(a1,v,a2),S)
                    == appl(seq(depiler-2, mov2(a1,v,a2)),S) ;

    appl(mov2(a1,v,a2),S)
                    == si v egal valeur '0'
                       alors S
                       sinon appl(seq(subst(cont(a2),cont(a1)),
                                      mov2(succ-ad(a1),pred(v),
                                           succ-ad(a2))),S)
                       finsi ;

%pour les ensembles :%

    appl(adj-1(v1),S) == appl(seq(depiler,adj-2(v)),S) ;
    appl(adj-2(v1),S) == si v egal valeur '0' alors S
                       sinon appl(seq(empiler(valeur '0'),
                                      adj-2(pred(v))),S)
                       finsi ;

    appl(srs-1(u,v),S)
                    == si u sup-val v egal logique 'vrai'
                       alors appl(seq(depiler-2, empiler(valeur '0')),
                                  S)
                       sinon appl(seq(depiler-2, srs-2(u,v,valeur '0')),
                                  S)
                       finsi ;

    appl(srs-2(u,v,n),S)
                    == si (u sup-eg-val mult(succ(n),taille-mot))
                                  egal vrai
                       alors appl(seq(empiler(valeur '0'),
                                      srs-2(u,v,succ(n))),S)
                       sinon si div-ent(u,taille-mot)  egal
                                div-ent(v,taille-mot)
                             alors appl(seq(empiler
                                            (fixe-bit
                                               (mod(u,taille-mot),
                                               (mod(v,taille-mot)
```

```
                                        (empiler(succ(n))),S)
                                sinon appl(seq(empiler
                                                (fixe-bit
                                                   (mod(u,taille-mot),
                                                   (pred(taille-mot))),
                                               srs-2
                                                   (mult(succ(n),taille-mot),
                                                   v,succ(n))),S
                                finsi
                            finsi ;

      appl(uni-1(n,a1,a2),S)
                        == si n egal valeur '0'
                            alors appl(subst(SP,index(succ-ad(SP),
                                                      cont(SP))),S)
                            sinon appl(seq(subst(cont(a2),
                                                 ou(cont(a1),
                                                   cont(a2))),
                                           uni-1(pred(n),succ-ad(a1),
                                                 succ-ad(a2))),S)
                            finsi ;
appl(inn-1(a),S)
         == appl(seq(subst(cont(a),
                           test-bit(mod(cont(a),taille-mot),
                                    cont(index(succ-ad(SP),
                                               div-ent(cont(a),
                                                       taille-mot))))),
                     subst(SP,a)),S) ;

%
int-1 est definie de meme que uni-1 avec "et" au lieu de "ou", et dif-1
avec "etnon" au lieu de "ou"
%

      appl(eqs-1(a1,a2,v,a3),S)
                        == si v egal valeur '0'
                            alors appl(seq(subst(SP,a3),
                                           subst(cont(SP),
                                           valeur '1')),
                                       S)
                            sinsi(cont(a1) eg-val cont(a2))
                                  egal faux
                            alors appl(seq(subst(SP,a3),
                                           subst(cont(SP),
                                                 valeur '0')),
                                       S)
                            sinon appl(eqs-1(succ-ad(a1),succ-ad(a2),
                                             pred(v),a3),S)
                            finsi ;

      appl(leqs-1(a1,a2,v,a3),S)
                        ==  %
                            meme terme que pour equs-1 avec le test du
                            sinsi remplace par : incl-sur-n-bits
                            (pred(taille-mot),cont(a1),cont(a2)) =
```

```
                                valeur '0'
                                %

%
neqs-1 est definie de meme que eqs1 en echangeant valeur '0' et
valeur '1'
%

%pour les procedures :%

      appl(appel-proc(v,e,a),S)
          == appl(sEQ(subst(SP,index(SP,
                                     cont(index(UB,
                                                opp(pred(mult(v,
                                                    valeur '2')))))),

                      %
                      reservation de place sur la pile pour les
                      variables de v
                      %

                       subst(cont(SP),v),

                      %n. de proc%

                       empiler(e),

                      %retour de l'appelant%

                       empiler(MP),

                      %lien dynamique%

                       empiler(a),

                      %lien statique%

                       subst(MP,index(SP,valeur '3')),
                       ujp(eti 'v')),S) ;

      appl(rpu-1(v,e),S) == appl(sEQ(subst(MP,cont(index(MP,opp
                                                   (valeur '2')))),
                                     subst(SP,index(index(SP,
                                                    valeur '4'),v))),
                                     ujp(e)),S) ;

%Operations :%

%pour les acces, les adresses, les entiers, les booleens :%

     appl(rien,S) == S ;


%rangement de constante :%
```

```
      appl(sldc0,S)   == appl(ldci(valeur '0'),S) ;
      % .... %
      appl(sldc31,S)   == appl(ldci(valeur '31'),S) ;
      appl(ldcn,S)     == appl(empiler(nil),S) ;
      appl(ldcb(v),S)   == appl(ldci(v),S) ;
      appl(ldci(v),S) == appl(empiler(v),S) ;
      appl(lac(v),S)    == appl(empiler(index(CP,v)),S) ;

%acces local :%

      appl(sldl1,S)    == appl(ldl(valeur'1'),S) ;
      % ... %
      appl(sldl16,S)   == appl(ldl(valeur '16'),S) ;
      appl(ldl(v),S)   == appl(empiler(cont(index(MP,v))),S) ;
      appl(lla(v),S)   == appl(empiler(index(MP,v)),S) ;
      appl(stl(v),S)   == appl(ranger(index(MP,v)),S) ;

%acces global :%

      appl(sldo1,S)    == appl(ldo(valeur '1'),S) ;
      % .... %
      appl(sldo16,S)   == appl(ldo(valeur '16'),S) ;
      appl(ldo(v),S)   == appl(empiler(cont(index(BP,v))),S) ;
      appl(lao(v),S)   == appl(empiler(index(BP,v)),S) ;
      appl(sro(v),S)   == appl(ranger(index(BP,v)),S) ;

%Acces intermediaire :%

      appl(lod(v1,v2),S)  == appl(empiler(cont(index(enreg-ancetre(v1),
                                                v2))),S) ;
      appl(lda(v1,v2),S)  == appl(empiler(index(enreg-ancetre(v1),
                                                v2)),S) ;
      appl(str(v1,v2),S)  == appl(ranger(index(enreg-ancetre(v1),
                                                v2)),S) ;

%acces indirect :%

      appl(sind0,S)          == appl(ind(valeur '0'), S) ;
      % .... %
      appl(sind7,S)          == appl(ind(valeur '7'), S) ;
      appl(ind(v),S)         == appl(subst(cont(SP),cont(index(cont(SP),
                                                v))),S) ;
      appl(sto,S)            == appl(sec(ranger(cont(succ-ad(SP))),
                                        depiler),S) ;

%acces multiples :%

      appl(ldc(v1,v2),S)  == appl(empiler-mult(index(CP,add(v1,
                                 pred(v2))), v2),S) ;
      appl(ldm(v),S)         == appl(ldm1(index(cont(SP),pred(v)),v),S) ;
      appl(stm(v)S)          == appl(stm-1(cont(index(SP,v)),v),S) ;
      appl(mov(v),S)         == appl(mov-1(cont(SP),v,cont(succ-ad(SP))),S) ;

%calcul d'adresses :%
```

```
      appl(inc(v),S)        == appl(subst(cont(SP),index(cont(SP),v)),S) ;
      appl(ixa(v),S)        == appl(seq(subst(cont(succ-ad(SP)),
                                              index
                                                 (cont(succ-ad(SP)),
                                                  mult(cont(SP),v)))),S) ;

%Operations sur les booleens :%

      appl(bnot,S)          == appl(subst(cont(SP),non-bool(cont(SP))),S) ;

%Operations sur les entiers :%

      appl(abi,S)           == appl(subst(cont(SP),abs(cont(SP))),S) ;

%
ngi, inci, deci sont definies de meme en remplaçant abs par opp,
succ, pred%

      appl(adi,S)           == appl(seq(subst(cont(succ-ad(SP)),
                                              add(cont(succ-ad(SP)),
                                                  cont(SP))),
                                        depiler),S) ;

%
sbi, mpi, dvi, modi, equi, neqi, leqi, lesi, geqi, gtri
sont definies de meme en remplaçant add par
sous, mult, div-ent, mod, eg, non-eg, inf, inf-eg, sup,
sup-eg
%

      appl(chk,S)           == appl(depiler-2, S) ;

%Operations sur les ensembles :%

      appl(adj(v),S)     == si v egal cont(SP)
                            alors appl(depiler,S)
                            sinon appl(adj-1(cont(SP)),S)
                            finsi ;
      appl(srs,S)        == appl(srs-1(cont(succ-ad(SP)),
                                       cont(SP)),S) ;

      appl(inn,S) == appl(inn-1(index(succ-ad(SP),
                                      cont(SP))),S) ;

      appl(uni,S) == appl(uni-1(cont(SP), succ-ad(SP),
                                index(succ-ad-2(SP),cont(SP))),S) ;

%
int et dif sont definies de meme que uni avec int-1 et dif-1 au
lieu de uni-1
%

      appl(equs,S) == appl(equs-1(succ-ad(SP),
                                  index(succ-ad-2(SP),cont(SP)),
                                  cont(SP),
```

```
                                    index(index(succ-ad(SP),cont(SP)),
                                          cont(index(succ-ad(SP),
                                                     cont(SP)))),S) ;

%
neqs et leqs sont definis de meme que equs avec neqs-1 au lieu
de equs-1
%

%
geqs est definie de meme que leqs mais en inversant les 2 premiers
arguments de leqs-1
%

%les branchements :%

%la modif etiquetee et la sequence :%

      sortie(m,e) egal faux => appl(etiqueter(e,m),S) == appl(m,S) ;
      appl(seq(etiqueter(e,m)m'),S)  == appl(etiqueter(e,seq(m,m')),S) ;
      sortie(m,e) egal faux => appl(seq(m,m'),S) == appl(m',appl(m,S)) ;

%sauts en avant :%

      appl(sEQ(ujp(e),m1,etiqueter(e,m2)),S) == appl(etiqueter(e,m2),S) ;

%soit m = sEQ(fjp(e),m1,etiqueter(e,m2)) :%

      cont(SP) egal valeur '0' => appl(m,S) = appl(seq(depiler,
                                                       etiqueter(e,m2)),
                                                   S) ;

%
de meme pour tjp en remplaçant fjp par tjp et en echangeant 0
et 1
%

%
de meme pour eqj en remplaçant les axiomes par cont(SP) = cont par
(succ-ad(SP)) appart S et cont(SP) = cont(succ-ad(SP)) appart-pas S et depile
par depiler-2
%

%de meme pour neqj en inversant les deux definitions %

     appl(seq(cjp(v),m),S)
            == appl(seq(ujp(cont(index(index(CP,v),
                                       succ(sous(cont(SP),
                                                 cont(index(CP)),V))))),
                        m),S) ;

%
dans un terme objet  correct, cjp est suivi d'une suite d'instructions
etiquetees, terminees (sauf la derniere) par un saut inconditionnel a
la fin du case
```

```
%

%les boucles :%

%
soit m eg-eti etiqueter(boucle,SEQ(m1,tjp(ext),m2,ujp(boucle),
                                  etiqueter(ext,m3)))

avec pour tout e sortie(m1,e) = faux
     pour tout e sortie(m2,e) = faux
%

     (cont(SP) egal valeur '1' appart appl(m1,S)
          => appl(m,S) == appl(SEQ(m1,depiler,etiqueter(ext,m3)),S)
     (cont(SP) egal valeur '0' appart appl(m1,S)
          => appl(m,S) == appl(m,appl(SEQ(m1,depiler,m2),S))

%
de meme avec fjp au lieu de tjp en echangeant valeur '0' et
valeur '1'
%

%les sauts pour les procedures :%

     (sortie(m1,e) egal faux) et-log((e egal e3) appart appl(m1,S))

     => appl(sEQ(ujp(e1),etiqueter(e2,m2),ujp(e),
                 etiqueter(e1,m1),ujp(e2),
                 etiqueter(e3,m3)),S)

     == appl(sEQ(ujp(e1),etiqueter(e2,m2),ujp(e),
                 etiqueter(e1,m1),m2,
                 etiqueter(e3,m3)),S) ;

%les procedures :%

     appl(rpu(v),S) == appl(rpu-1(v,cont(pred-ad(MP))),S)

                       %
                       pred-ad(MP) contient l'etiquette retour.
                       v est le nb de mots a depiler
                       marque non comptee, soit
                       le nb de mots occupes par les variables
                       et les parametres.
                       avant le retour, la pile est vide, donc
                       MP = index(SP,valeur '3').
                       %

     appl(cpl(v,e),S)
                    == appl(appel-proc(v,e,MP),S) ;
     appl(cpg(v,e),S)
                    == appl(appel-proc(v,e,BP),S) ;
     appl(cpi(v1,v2,e),S)
                    == appl(appel-proc(v2,e,lien-stat(v1,MP)) ;
```

```
                                %
                                "cpl, cpg, cpi sont les appels de proc.
                                locales, globales ou intermediaires, ne
                                different que par le lien statique a mettre"
                                "e est une etiquette de retour qui sera
                                fournie par le compilateur. Les compilateurs
                                font parfois cela, bien que le langage
                                objet possede un compteur ordinal.
                                Nous suivrons cette idee qui permet une
                                specification claire"
                                %

    preconditions :
    %**************%


%pour  les acces, les adresses, les entiers, les booleens :%

      pre(ldcb(v)) == octet(v) ;
      pre(ldci(v)) == mot(v) ;
      pre(lac(v))  == mot-pos(v) ;
      pre(ldl(v))  == mot-pos(v) ;

% de meme pour lla(v), stl(v), lds(v), lao(v), sro(v)%

      pre(lod(v1,v2))     == octet-pos(v1) et-log mot-pos(v2) ;

      %de meme pour lda, str%

      pre(ind(v))   == mot-pos(v) ;
      pre(ldc(v1,v2))
                    == mot-pos(v1) et-log octet-pos(v2) ;
      pre(empiler-mult(a,v))
                    == octet-pos(v) ;
      pre(ldm(v))   == octet-pos(v) ;
      pre (stm(v))  == octet-pos(v) ;
      pre(mov(v))   == mot-pos(v) ;
      pre(ldm1(a,v))== octet-pos(v) ;

      %de meme pour stm1 et stm2%

      pre(mov1(a1,v,a2))
                    == mot-pos(v) ;
      pre(mov2(a1,v,a2))
                    == mot-pos(v) ;

%les adresses :%

      pre(inc(v))   == mot-pos(v) ;
      pre(ixa(v))   == mot-pos(v) ;

%les booleens :%

      pre(bnot)     == (cont(SP) eg-val valeur '0') ou-log(cont(SP)
                                eg-val valeur '1') ;
```

```
%les entiers :%

      pre(abi)       == mot(cont(SP)) et-log non-log(SP) eg-val
                                      valeur-min) ;

      %meme precondition pour ngi et pour deci%

      pre(inci)      == mot(cont(SP)) et-log non-log (cont(SP) eg-val
                                      valeur-max) ;
      pre(adi)       == mot(cont(SP)) et-log mot(cont(succ-ad(SP))) ;

%
de meme pour sbi, mpi, dir, modi, equi, neqi, leqi, lesi, geqi,
gtri
%

      pre(modi)      == cont(SP) sup-val valeur '0' ;
      pre(divi)      == non-log (cont(SP) eg-val valeur '0') ;

    cas d'echec :
    %***********%


%les entiers :%

      (cont(succ-ad-2(SP)) sup-val cont(SP)) ou-log
      (cont(succ-ad-2(SP)) inf-val (cont(succ-ad(SP)))
                           => echec(chk) ;


%les ensembles :%

      non-log octet(v) => echec(adj(v)) ;
      (cont(SP) inf-val valeur '0') ou-log (cont(SP) sup-val card1)
                                      => echec(srs) ;
      (cont(succ-ad(SP)) inf-val valeur '0') ou-log
      (cont(succ-ad(SP)) sup-val card1)      => echec(srs) ;
      (cont(SP) inf-val valeur '0' ou-log cont(SP) sup-val card1)
                                      => echec(inn) ;
      non-log(cont(SP) eg-val cont(index(SP,succ(cont(SP)))))
                                      => echec(uni) ;

  %de meme pour int, dif, equs, neqs, leqs, geqs%

  %les branchements :%

  %fjp et tjp echouent si il n'y a pas un booleen au sommet de pile :%

      non-log(cont(SP) eg-val valeur '0')
      et-log non-log(cont(SP) eg-val
                     valeur '1')
                       => echec(seq(fjp(e),n1,etiqueter(e,m2))) ;

  %de meme pour tjp%
```

```
%
cjp(r) echoue si l'index du sommet de pile ne pointe pas dans la zone
d'etiquette allouee a ce case :%

      (cont(SP) inf-eg-val cont(index(CP,v))) ou-log
      (cont(SP) sup-eg-val cont(index(CP,succ(v))))
                           => echec(seq(cjp(v),m)) ;

%
cjp(v) echoue si l'etiquette a laquelle on doit se brancher n'apparaît
pas dans la suite du programme :%
%

      non-log entree(m,cont(index(index(CP,v),
                                  succ(sous(cont(SP),
                                            cont(index(CP,v))))))
                        => echec(seq(cjp(v),m)) ;

%l'echec optionnel :%

      echec-deb(empiler(m)) => SP eg-ad succ-ad(NP) ;
      echec-deb(appel-proc(v,e,a))
                           => SP eg-val succ-ad(NP) ;

%les procedures :%

      non-log (octet-pos(v)) => echec(cpl(v,e)) ;

%de meme pour cpg, cpp%

      (non-log (octet-pos(v1))) ou-log (non-log (octet-pos(v2)))
                          => echec(cpi(v1,v2,e)) ;
      non-log (mot-pos(v))    => echec(rpu(v)) ;
      non-log (octet-pos(v)) => echec(appel-proc(v,e,a)) ;
      non-log (mot-pos(v))    => echec(rpu-1(v,e)) ;

fin modif ;
%**********%
```

A 3. - Index dans le type abstrait donné dans l'annexe A.2.

- Type logique :

Opérations	Déclarée page	Axiomatisée page
vrai	1	1
faux	1	1
non-log	1	1
et-log	1	1
ou-log	1	1
eg-log	1	1
inf-log	1	1
sup-log	1	1

- Type mot :

Opération	Déclarée page	Axiomatisée page
cont	2	2

- Type adresse :

Opérations	Déclarée page	Axiomatisée page
LB	2	
NB	2	
CP	2	
NIL	2	
UB	2	
SP(mod)	2	
MP(mod)	2	
BP(mod)	2	
succ-ad	2	
succ-ad-2	2	2
pred-ad	2	2
index	2	2

enreg-ancêtre	2	3
enreg-ancêtre-1	2	3
eg-ad	2	3
inf-ad	2	3
sup-ad	2	3

- Type valeur :

Opérations	Déclarée page	Axiomatisée page
valeur-min	4	
valeur-max	4	
succ	4	4
valeur-octet-min	4	
valeur-octet-max	4	
pred	4	5
opp	4	5
abs	4	5
non-bool	4	5
exp-2	4	5
add	4	5
sous	4	5
mult	4	5
div-ent	4	6
mod	4	6
eg	4	7
non-eg	4	7
inf	4	7
inf-eg	4	7
sup	4	7
sup-eg	4	7
eg-val	4	7
inf-eg-val	4	7
inf-val	4	7
sup-val	4	7
sup-eg-val	4	7

zéro	4	
pred-valeur	4	6
succ-valeur	4	6
opp-valeur	4	6
add-valeur	4	6
mult-valeur	4	6
inf-eg-valeur	4	6
mot	4	8
octet	4	8
mot-pos	4	8
octet-pos	4	8
taille-mot		
card-1		
non	4	8
inv-bit-de-signe	4	8
ou	4	8
et	4	8
test-bit	4	8
fixe-bit	4	9
incl-sur-n-bits	4	9
lien-stat	4	9

- Type_eti :

Opérations	Déclarée page	Axiomatisée page
étiqueter	10	19
entrée	10	11,12
sortie	10	11,12
eg-eti	10	11

- Type modif

Opérations	Déclarée page	Axiomatisée page
Opérations cachées :		
pour les accès, les adresses, les entiers, le booléen :		
dépiler	12	13
dépiler-2	12	13
empiler	12	13
ranger	12	13
empiler-mult	12	13
ldm-1	12	14
stm-1	12	14
stm-2	12	14
mov-1	12	14
mov-2	12	14
pour les ensembles :		
adj-1	12	14
adj-2	12	14
uni-1	12	14
int-1	12	14
srs-1	12	14
srs-2	12	14
equs-1	12	15
neqs-1	12	16
leqs-1	12	15
inn-1	12	15
pour les procédures :		
appel-proc	12	16
rpu-1	12	16

Opérations

pour les accès, les adresses, les entiers et les booléens :

rien	12	16
seq	12	19

rangement de constante

sldc0	12	17
sldc31	12	17
ldcn	12	17
ldcb	13	17
ldci	13	17
lac	13	17

accès local :

sldl1	12	17
sldl16	12	17
ldl	13	17
lla	13	17
stl	13	17

accès global :

sldo1	12	17
sldo16	12	17
ldo	13	17
lao	13	17
sro	13	17

Accès intermédiaire

lod	13	17
lda	13	17
str	13	17

Accès indirect

sind0	12	17
sind7	12	17
ind	13	17
sto	13	17

Accès multiple :

ldc	13	17
ldm	13	17
stm	13	17
mov	13	17

Calcul d'adresse :

inc	13	18
ixa	13	18

Opérations sur le booléens :

bnot	12	18

Opération sur les entiers :

abi	12	18
ngi	12	18
inci	12	18
déci	12	18
adi	12	18
sbi	12	18

mpi	12	18
dvi	12	18
modi	12	18
equi	12	18
neqi	12	18
leqi	12	18
lesi	12	18
geqi	12	18
gtri	12	18
chk	12	18

Opérations sur les ensembles :

adj	13	18
srs	13	18
inn	13	18
uni	13	18
int	13	18
dif	13	18
equs	13	18
neqs	13	18
leqs	13	18
geqs	13	18

les branchements :

ujp	13	19
fjp	13	19
tjp	13	19
eqj	13	19
neqj	13	19
cjp	13	19

BIBLIOGRAPHIE

AJN 75 — The Pascal (P) Compiler : implementation notes.
K.V. Nori , U. Amman, K. Jensen, H.H. Nägeli
Eidgenossische Technische Hochschule - Zürich - 1975 -

Des 80 — Production de compilateurs à partir d'une description sémantique des langages de programmation : le système Perluette
Ph. Deschamp - thèse de docteur-ingénieur - 1980 -

Gau 80 — Génération et preuve de compilateurs basées sur une sémantique formelle des langages de programmation.
M.C. Gaudel - thèse d'état - 1980 -

GTW 78 — Abstract data types as initial algebras and the correctness of data representations. Goguen - Thatcher Wagner current trends in Programming Methodology - 1978 - p. 80 149 -

ISO 81 — Norme iso de Pascal - ISO/DIS 7185

JKM 74 — A Pascal environment machine (P-Code)
B.B. Kristensen, O.L. Madsen, B.B. Jensen
DAIMI PB-28 - 1974 -

JW 78 — User manual and report
K. Jensen, W. Wirth
Springer Verlag (Sd. Corrected Reprint of the sd. ed.)
- 1978 -

Mor 76 — A manual for MODEL
J.B. Morris
A lamos, New Mexico - 1976 -

Nel 79 — A comparison of Pascal intermediate languages
Philip A. Nelson
Lawrence Livermire Laboratory - University of California -
Davis - 1979 -

Use 81 — UCSD P-system and UCSD Pascal - Version IV.0
User's manual - Sd. Ed. January 1981 -
Sortech Microsystems, INC. San Diego - 1981 -

Imprimé en France

par

l'Institut National de Recherche en Informatique et en Automatique